The Design and Implementation of a Load Distribution Facility on Mach

by

HSIEH Shing Leung Arthur

A thesis submitted in partial fulfilment of the
requirements for the degree of
Master of Philosophy

in the

Division of Computer Science and Engineering
Graduate School
of
The Chinese University of Hong Kong

June 1997

Abstract

The Design and Implementation of a Load Distribution Facility on Mach

by

HSIEH Shing Leung Arthur
Master of Philosophy

The Chinese University of Hong Kong

A variety of approaches have been used in the study of load distribution (LD). They include queuing network analysis, simulation, experimental implementation and measurements. Most studies employ the approaches of queuing network analysis or simulation. Much less work has been done to actually implement LD algorithms for practical use. Only few projects actually designed and implemented LD facilities for their studies but none of them offer their facilities as convenient tools to support dynamic replacement of different LD algorithms.

In this thesis, we present the architecture, design and implementation issues of the Distributed Scheduling Framework (DSF), a LD tool built on top of the Mach operating system to support distributed scheduling. It extends the scheduling capability of the Mach kernel and allows on-line replacement of LD algorithms without the need to shut down the system. It also supports multiple LD algorithms running on the same host concurrently so that users can choose a suitable LD algorithm for submitting their tasks according to their need. The number of hosts involved in LD scheme provided by DSF can be changed dynamically. A host can join and leave our LD scheme without needing to restart the whole DSF system. DSF is completely transparent to tasks submitted to it for distributed scheduling. Services provided by DSF make implementation of LD algorithms much easier. DSF also provides a test bed for verifying new LD algorithm's performance with its theoretical and simulation results.

A preliminary study of some well-known LD algorithms by using our DSF is presented. The result shows that LD with sender-initiated algorithms under homogeneous workload is desirable when system load is low to moderate. At high system load, LD activities should be avoided. For heterogeneous workload condition, LD can significantly

improve system performance. When multiple LD algorithms are used for distributed scheduling, we got generally averaged performance from component algorithms.

Acknowledgements

I wish to express my deep gratitude to Dr. Qin Lu, my research supervisor, for her supervision during this two years. Her constant encouragement and advice help me a lot to accomplish this research.

I wish to thank Dr. John Lui and Dr. Ada Fu for acting as Internal Examiners of my Thesis Committee.

I also wish to thank Mr. Sau-Ming Lau for giving me a lot of valuable advice on my experimental studies. He is always willing to spend his time with me on discussing the problems encountered in my experiments.

I wish to thank Mr. Peter Wai-kwok Lie for his help during the early stage of this research.

Finally, I gratefully appreciate the technical support from Mr. Angus Siu in answering my questions about CMU Mach system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a distributed system which consists of a set of nodes connected by a local area network, it is very likely that nodes have uneven workload because of statistical fluctuations in the arrival of tasks to nodes and task service time requirement [23]. Load distribution (LD) aims at improving the system performance by transferring some of the workload from a heavily loaded node to other lightly loaded or idle nodes. The average response time of tasks is usually chosen as a system performance measure. Thus the goal of LD is trying to minimize the average response time.

Load distribution has been studied extensively because it can offer many potential benefits like resource sharing, fault tolerance and improving real-time characteristics [20]. A variety of approaches has been used in the study of load distribution. They include queuing network analysis, simulation, experimental implementation and measurements [32]. Most studies employ the approaches of queuing network analysis or simulation. Much less work has been done to actually implement LD algorithms for practical use. The reasons for the lack of work in this area are three-fold. Firstly, studies in [9, 23] have shown that no single LD algorithm outperforms others in all situations; for example, Eager *et al.* [9] showed that sender-initiated algorithms outperform receiver-initiated algorithms at light to moderate system loads but the reverse is true at high system loads. It is desirable that different systems can use different LD algorithms. Even in the same system, the necessity of changing a LD algorithm may arise when the system configuration or other system parameters, such as task types, are changed. Secondly, changing LD algorithms in a scheduler is a non-trivial task. A scheduler using a LD algorithm is usually a system protected object. Any modification or replacement of LD algorithms thus involves at least reconfiguration and restart of the

system. Thirdly, a LD facility is needed in experimental and measurement studies and implementing such a facility in a distributed system requires a substantial effort and is obviously difficult [34].

On the other hand, only few projects, such as MOSIX [2], Utopia [33], SAHAYOG [7] and others in [20, 34, 27, 32], actually designed and implemented a LD facility for their studies. However, they do not provide a convenient mechanism to support replacement of different LD algorithms. Besides, most of these systems do not provide the LD facility which is completely transparent to application programs. For instance, applications can only benefit from LD offered by Utopia [33] if they are modified to use its interface library. This makes these application programs less portable to other systems. Thus it is desirable that ordinary application programs can enjoy load distribution without recompilation or changes in their source program.

With a system tool that is flexible enough to allow different LD algorithms to be loaded and used, LD algorithms can be actually used to improve system performance. Besides, such a tool can be used as a test bed to verify new algorithm's actual performance with its theoretical or simulation results.

In this thesis, we present the Distributed Scheduling Framework (DSF) which is a LD tool that supports distributed scheduling by using "plugged in" LD algorithms. It extends the scheduling capability of the Mach kernel and allows on-line replacement of LD algorithms at the user level without any kernel reconfiguration or system restart. It also supports multiple LD algorithms running on the same host concurrently so that users can choose a suitable LD algorithm for submitting their tasks according to their need. DSF also facilitates the implementation of LD algorithms by providing services commonly required by the algorithms. Furthermore, DSF is transparent to application programs. Thus most of the existing application programs can immediately benefit from LD provided by DSF without the need of source code modification or recompilation.

The Mach operating system [1] was chosen as the platform to implement the DSF because, being a microkernel, Mach allows system services to be provided at the user level. The support of multiple threads of control within a single address space and Mach's particular type of interprocess communication (IPC) are essential to DSF implementation. Since Mach has been ported to different hardware architectures, it is also possible to extend DSF to a heterogeneous distributed computing environment.

The rest of the thesis is organized in the following chapters. In Chapter 2, we give a review on some important elements in load distribution. An overview of important fea-

tures of Mach, our implementation platform, is presented and a survey on some research projects is also given. In Chapter 3, we describe our system model and design objectives in building DSF. The architecture of DSF is explained and the design issues is discussed thoroughly. In Chapter 4, we explain the implementation details of building DSF. Major techniques used in Mach programming environment are described and some peculiar problems encountered are also discussed. In Chapter 5, we present our experimental results on studying some well-known LD algorithms by using our DSF. Performance of using multiple LD algorithms for load distribution is also compared with its component algorithms. Finally, we give our conclusion and direction of future work in Chapter 6.

Chapter 2

Background and Related Work

In load distribution, we can usually separate the part concerning policy from the part concerning mechanisms. The policy refers to the role played by load distribution algorithm. In contrast, the mechanisms refer to the work, like collecting load information and transferring a task, that is needed by the policy to fulfill the load distribution objective. In this chapter, we explain the elements involved in load distribution mechanisms and algorithm.

2.1 Load Distribution

In [20], Milojicic gives an informal definition of Load Distribution (LD) as follows

> *LD is a common name for various techniques that perform transfer of computing load between the nodes in a distributed system, in order to achieve a desired goal.*

In fact, there are various terminologies used in this field also having the similar meaning to load distribution. The most frequently encountered ones include load balancing, load sharing and distributed scheduling. Load balancing has the objective of equalizing the workload among nodes [12, 32] or balancing the queue lengths at the nodes [8]. Load sharing attempts to conserve the ability of the system to perform work by assuring that no node is idle while processes wait for service [12, 8], or it means a redistribution of the workload [32]. Distributed scheduling can be regarded as comprising two components. The *local scheduling* addresses the allocation of processing resources to jobs within one node. The *global scheduling* determines that which jobs should be processed by which

node [28]. In this thesis, we use the term load distribution to collectively refer to load balancing, load sharing or distributed scheduling.

2.1.1 Load Index

A *load index* is a quantitative measure of the load on a host [30]. It is used by LD algorithm to make task transfer decision since it predicts the performance of a task if it is executed at some particular node. An effective load index should have good correlation between task response time and its value during task submission. It is crucial that the mechanism for measuring load is efficient and incurs less overhead.

Different kinds of load indexes have been proposed. They include the CPU queue length, the average of CPU queue length over some period, the amount of available memory and CPU utilization [23]. Zhou in [29] experimentally validated the suitability of the resource queue lengths as load indexes. He found that the CPU and disk queue lengths have very high correlation to the response times of CPU and I/O bound jobs respectively.

2.1.2 Task Transfer Mechanism

Task (or process) transfer is a mechanism used by LD algorithm for distributing system workload among nodes in a distributed system. Task transfer can be performed either non-preemptively or preemptively [12, 23]. The non-preemptive task transfer is called task placement and it involves only tasks that have not started execution. The task placement works by selecting a suitable node as the execution site and initiating a task at that node. In contrast, the preemptive task transfer, also known as task migration, involves transferring a partially executed task to another node for further execution.

In general, task migration is more costly than task placement because it needs to collect a task's state which can be quite complex after the task's execution has started. A task state usually includes a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, and others [23]. The mechanism also involves suspending the currently executing task, collecting its execution state information, transferring the state information to another node and resuming the execution of the suspended task. Although task migration provides the flexibility that the execution of a task is not restricted to a particular node, the overhead of providing

such flexibility may be too high that it outweighs its provided performance improvement to the system.

2.1.3 Load Distribution Facility

A LD facility is used to realize the potential benefit brought by load distribution in a distributed system. It consists of both policy and mechanisms of load distribution. Mechanisms usually refer to load information collection and task transfer. Policy means the part served by LD algorithm. This facility monitors variation in system load to detect load imbalance, and then takes action to balance the load [35].

2.2 Load Distribution Algorithm

Load distribution algorithm is an important component in load distribution because it decides which task should be allocated to which node for processing. In this section, we discuss several important aspects about LD algorithms.

2.2.1 Classification

Load distribution algorithms can be broadly classified into three types. They are static, dynamic and adaptive. LD algorithms that use information about the average behavior of the system but ignore the current system state are regarded as static algorithms. They may be either deterministic or probabilistic. Static algorithms assume that information about the average execution times and intercommunication requirements of all tasks are known. Their goal is to deterministically allocate tasks to nodes so that the total time to process all tasks is minimized [8]. Dynamic LD algorithms use system state information (system loads) to make the decision on allocating tasks to other nodes. They collect and react to system state and thus they are more complex and having more overhead than static algorithms. Dynamic algorithms improve system performance by exploiting short-term fluctuation of system state. Adaptive LD algorithms can be regarded as a special class of dynamic LD algorithms. When the algorithms are used, both system load and performance are constantly monitored, and changes in algorithms and/or adjustable parameters are made automatically as the load changes so that the system is always operating at, or close to, its optimal point [31, 23].

Dynamic LD algorithms can be further classified according to their degree of centralization. In this scheme, LD algorithm can be centralized, hierarchical, distributed, or

some combination of these. The centralized algorithm makes LD decision from a single node. It is potentially less reliable than the distributed counterpart because failure in the node that has the centralized algorithm running can make the whole LD system fail. In addition, the centralized algorithm can easily become the bottleneck of the LD system when the number of nodes involved is large. In contrast, all nodes are involved in making LD decision in the distributed algorithm. Hierarchical algorithm combines the good characteristics of distributed and centralized algorithms. It assumes that the distributed system is organized into clusters of a small number of nodes. The distributed algorithm is used within a cluster and the centralized algorithm is used among clusters [20].

The dynamic LD algorithm with distributed control can be further classified into the following three types—sender, receiver and symmetrically initiated. In sender-initiated (or source-initiative) algorithms, LD activity is initiated by an overloaded node which tries to send a task to an underloaded node. Queues tend to form at the sender node and LD decision is usually made at job arrival time. In receiver-initiated (or server-initiative) algorithms, LD activity is initiated from an underloaded node, which tries to get a task from an overloaded node. Queues tend to form at the nodes other than the receiver and LD decision is usually made at job departure time [23, 28, 9]. Under symmetrically initiated algorithms, a node initiates a LD activity for task transfer when it is either overloaded or underloaded. These algorithms have the advantages of both sender-initiated and receiver-initiated algorithms. When system load is low, the sender-initiated component is more successful in finding underloaded nodes. When system load is high, the receiver-initiated component is more successful in finding overloaded nodes [23].

Figure 2.1 summarizes the classification of LD algorithms that we have described. In this thesis, we will mostly focus our discussion on LD algorithms that are dynamic, distributed and sender-initiated.

2.2.2 Components

A typical dynamic LD algorithm consists of a transfer policy, a selection policy, a location policy and an information policy [23].

Transfer policy. A transfer policy determines whether a node is in a suitable state to participate in a task transfer, either as a sender or a receiver. *Threshold* policy [8, 9] is the most commonly used transfer policy. Threshold is expressed in terms of load unit. When a new task originates at a node, the transfer policy determines that the node is

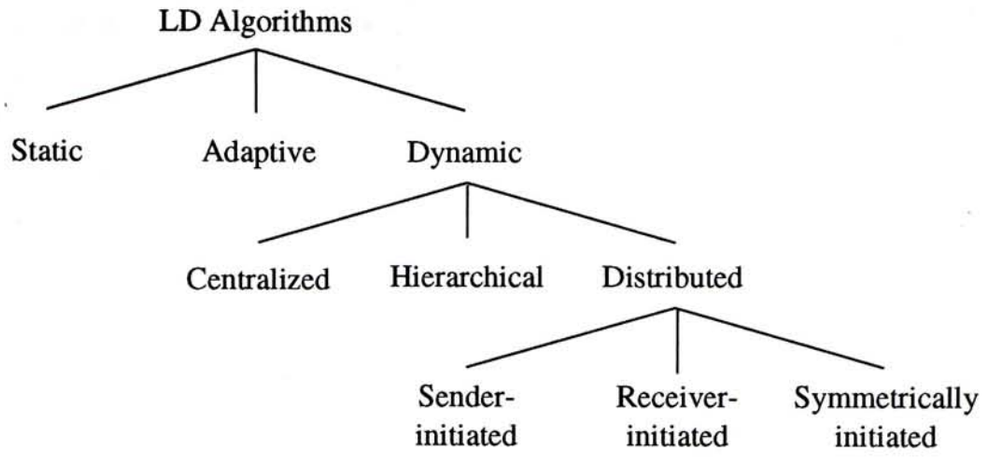

Figure 2.1: Classification of LD algorithms.

a sender if the load at that node exceeds a threshold T_1. On the other hand, if the load at a node is below T_2, the transfer policy decides that the node can be a receiver for a remote task.

Selection policy. Once the transfer policy decides that a node is a sender, a selection policy is used to choose a task for transfer. A simple approach is to select one of the newly originated tasks that caused the node to become a sender. Such a task is a good candidate for task transfer because the transfer is non-preemptive. When selecting a task, the following factors should be considered:

- The overhead incurred by the transfer should be minimal.
- The selected task should be long running so that the transfer overhead is small when comparing its remaining resource requirement.
- The number of *location-dependent* system calls made by the selected task should be minimal. Location-dependent system calls are those that must be executed in the originated node.

Location policy. The location policy is responsible for finding a suitable node (sender or receiver) for task transfer. A commonly used location policy in distributed algorithms is *polling*. In this policy, a node polls another node to find out whether it is suitable for load distribution. A node can be selected for polling on a random basis, or on the information collected from its information policy.

Information policy. The information policy decides when information about the states of other nodes in the system is to be collected, from where it is to be collected, and what information is collected. The following are three common information policies:

1. *Demand-driven policy.* In this policy, a node collects the state of other nodes only when it becomes either a sender or a receiver, making it a suitable candidate to initiate load distribution. Demand-driven policy can be used in sender, receiver, or symmetrically initiated algorithms.

2. *Periodic policy.* This policy can be used in either centralized or distributed algorithms to collect information periodically. Periodic policy does not adapt its rate of activity to the system state. For instance, under high system load, there is not much gain from performing LD activity. However, the overhead from periodic information collection can further increase the system load and therefore even worsen the situation.

3. *State-change-driven policy.* In this policy, nodes disseminate information about their states whenever their states change by a certain degree. This policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. In centralized algorithms, nodes send state information to a centralized collection point whereas nodes send information to peers in distributed algorithms.

2.2.3 Stability and Effectiveness

The stability of LD algorithms can be viewed from the queuing theoretic and algorithmic perspectives [23]. In queuing theory, when the long-term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound and the system is regarded as unstable. Suppose a system is originally stable, we can have the following cases when a LD algorithm is used to improve the system performance:

1. The whole system is stable and the system performance is improved.

2. The system performance is worse than the one without using LD algorithm but the whole system is still stable.

3. The whole system becomes unstable.

We call the LD algorithm to be *unstable* when the third situation occurs. It happens because the total load resulted from the external arriving jobs and the overhead imposed by the algorithm, such as performing message exchanges to collect state information,

exceeds the service capacity of the system. In the first and the second cases, the LD algorithm does not trigger the system into an unstable state. However, in the first case, the LD algorithm is regarded as *effective* because it actually improves the system performance. In the second case, the LD activities do not outperform the overhead introduced by the algorithm itself.

In algorithmic perspective, if an algorithm can perform fruitless actions indefinitely with nonzero probability, the algorithm is unstable. For example, in processor thrashing, the transfer of a task to a receiver may increase the receiver's queue length to the point of overloading it, requiring the transfer of that task to another node. This process may repeat indefinitely. In this case, a task moves from one node to another in search of a lightly loaded node without ever receiving any service.

2.3 The Mach Operating System

Microkernels are kernels that provide the smallest possible set of services and resources on which the remaining services required can be built. The basic set of services includes a process abstraction, a memory management service and a basic communication service [6]. Mach [1] is a multiprocessor operating system kernel and environment developed at Carnegie Mellon University (CMU). It is an example of using the microkernel approach in its operating system design [3]. Mach kernel only provides process management, memory management, communication and I/O services. Files, directories, and other traditional operating system services are handled in user space [26].

2.3.1 Mach kernel abstractions

Mach kernel supports the following abstractions [1, 6, 15]:

- A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, a collection of kernel-managed capabilities for accessing ports and virtual memory).

- A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads within a task share access to all task resources.

- A *port* is a unicast, unidirectional communication channel with a message queue protected by the kernel. Ports are the reference objects in Mach. *Send* and *Receive* are the fundamental primitive operations on ports.

- A *port set* is a collection of port receive rights in a task. It is used to receive a message from any one of a collection of ports.

- A *message* is a typed collection of data objects used in communication between threads. Message may be of any size and may contain pointers and typed capabilities for ports.

- A *memory object* is an internal unit of memory management. It corresponds to a region of the virtual address space of a task. It is an object which is accessed by the kernel to perform virtual memory paging operations.

2.3.2 Mach kernel features

Mach provides the following key features [3, 22]:

- Task and thread management

  Mach supports the task and thread abstractions for managing execution. Computation within a task is performed by one or more threads which share the address space and all other resources of the task. Threads are scheduled to processors by the Mach kernel, and may run in parallel on a multiprocessor.

- Interprocess communication

  Mach provides interprocess communication (IPC) among threads via constructs called ports. Ports are protected by a capability mechanism so that only tasks with appropriate send or receive capabilities can access a port. All services, resources, and facilities within the Mach kernel, as well as those exported by particular Mach tasks or servers are represented as ports. Mach tasks, threads, memory objects, and processors are all manipulated by sending messages to ports which represent them. It can also be transparently extended over a network by using external communication servers.

- Memory object management

  The address space of a Mach task is represented as a collection of mappings from linear addresses to offsets within Mach memory objects. The role of kernel is

to manage physical memory as a cache of the contents of memory objects. The kernel's representation for the backing storage of a memory object is a Mach port to which messages can be sent requesting or transmitting memory object data.

- Integrated IPC and virtual memory management support

  Interprocess communication and memory management in Mach are tightly integrated. Memory management techniques, like copy-on-write, are employed whenever large amount of data are sent in a message from one program to another. Thus the transmission of megabytes of data can be at very low cost without actual data copying. Mach virtual memory objects are represented as ports. When a page fault occurs, the kernel sends a message to the backing storage port of a memory object to get the data contained in the faulted page.

- System call redirection

  The Mach kernel allows a designated set of system calls or traps to be handled by code running in user mode within the calling task. The set of emulated system calls needs to be set up only once; it is inherited by child tasks on fork operations. This feature allows the binary emulation of operating system environments such as Unix. It also allows for monitoring, debugging, and transparent extension of existing operating system functions.

- User multiprocessing

  A user-level multithreading package, the C thread library [5], facilities the use of multiple threads within an address space. It exports mutual exclusion mutex locks and condition variables for synchronization via `condition_wait` and `condition_signal` operations.

2.4 Related Work

Since the research area of load distribution has been studied for a long time, a lot of literature in this area has accumulated. Instead of giving an extensive discussion about every aspect of load distribution, we focus on the researches on the design and implementation of LD facility.

Zhou and Ferrari [32] implemented a prototype load balancer for a loosely-coupled distributed system for conducting measurement experiments. Their system is based

on a modified C shell for the Berkeley UNIX 4.3 BSD system running on DEC VAX machines. The modified C shell intercepts user commands and executes certain types of commands remotely when the local host is heavily loaded, using the `rexec` daemon provided by the system. When an eligible job is submitted by the user to the C shell, the C shell contacts a Load Information Manager (LIM) which constantly monitors the load of a host in the system and performs job placements. If the initiated host is heavily loaded, a remote underloaded host is selected and the placement decision is returned to the C shell. For remote execution, the C shell contacts the Load Balancing Manager (LBM) on the destination host, which starts up an R-shell and establishes a stream connection between it and the home C shell.

Zhou *et al.* [33] implemented a load sharing facility for large, heterogeneous distributed computer systems called Utopia. It is based on the clustering nature of large-scale distributed systems. Centralized algorithms are used within host clusters and directed graph algorithms are used among clusters for managing resource load information and task placement. It is scalable to thousands of hosts. Utopia clearly separates the policies governing the exchange of load information and task placement decision making from the mechanisms for transparent remote execution. It has two servers running in each host. The first server Load Information Manager (LIM) is the policy module of Utopia. LIM exchanges load information with its peers on other hosts and gives advice to applications as to on which host their task should be executed. The second server is Remote Execution Server (RES). RES provides the mechanisms for transparent remote execution of arbitrary tasks. Before remote task initiation, a stream connection is established between the local application and its remote task, through the RES on the target host.

SAHAYOG is a test bed for evaluating dynamic load-sharing policies [7]. The test bed was implemented on a network of AT&T 3B2 minicomputers. It provides an interactive user interface of conducting load-sharing experiments. Based on user-specified parameters, it creates independent job streams at different nodes in the network. Each node collects data about the jobs for generating statistics about the experiment. A special feature of SAHAYOG is that it contains an optional fault-tolerance feature to handle single-node failures, and evaluates the effect of fault tolerance on the performance of different policies.

Tsai *et al.* [27] implemented a load balance facility in the Distributed MINIX system. Both process migration and remote execution are mechanisms used for task transfer.

They also design a simple and efficient method to get the workload of a computer and a method to get process characteristics.

Zhu [34] implemented a load balancing facility on the Amoeba system for carrying out experiments to study various LD algorithms. The load balancing facility consists of two components: load balancer and process migration server. A load balancer makes load balancing decisions based on system states, and a migration server carries out these decisions to move processes between computers. Therefore, the load balancer plays the policy role and is responsible for gathering load information, identifying system state and determining actions. In contrast, the process migration server supplies the mechanism to implement decisions in a load balancing facility.

Milojicic *et al.* [21, 20] provides load distribution on top of Mach. The LD scheme consists of three major components: task migration, load information management and distributed scheduling. Task migration deals with the Mach task abstraction and leaving the OS personality abstraction (e.g. UNIX process) on the source machine. Two kinds of task migration servers were implemented in user space with some kernel modifications to provide task migration mechanism. Load information management includes information collection, dissemination and negotiation. Besides the traditional processing information, the load information also includes information on virtual memory and interprocess communication. Distributed scheduling is provided by running a user level program (LD server) on every node in a cluster of hosts. The nodes can join or leave the cluster at any time. The LD server supports five types of LD strategies—no LD, random, sender initiated, receiver initiated and symmetrical. The LD server periodically inspects the load on the local node using load information management interface. If the local load is above a threshold value, the LD server acts according to the currently active strategy. Although the LD server supports several LD strategies, its design does not aim at making LD strategy (LD policy) as a replaceable component in its LD scheme. Therefore, adding any new strategy will require modification on the LD server.

PVM supports programming of parallel applications on distributed systems but it does not provide mechanisms for assigning and migrating workload among processes of parallel programs during execution. Gold and Schnekenburger [10] added an adaptive LD system named ALDY to PVM. ALDY is a library of functions that provides mechanisms for load distribution and a collection of LD strategies. The approach used by PVM and ALDY requires applications to use its provided library interface for gaining parallelism and load distribution functionality. This is less desirable than our approach

of providing load distribution without modifying the applications.

Chapter 3

The Design of Distributed Scheduling Framework

The Distributed Scheduling Framework (DSF) is a load distribution tool which allows LD algorithms to be replaced easily. It supports distributed scheduling by using multiple LD algorithms together. It also facilitates the implementation of LD algorithm by providing a library of LD primitives. In this chapter, we discuss the architecture and design issues of components inside DSF.

3.1 System Model

We assume that our distributed system consists of a set of autonomous hosts connected by a local area network with each host running the same operating system. A DSF server runs on every host to participate in global scheduling and the operating system of each host is responsible for local scheduling.

We further assume that a LD algorithm has the following general behavior. It collects load information from some or all hosts in the distributed system. Whenever a task is submitted to the algorithm, it first determines whether to schedule the task locally or remotely (transfer policy) for execution. If the remote assignment of the task is needed, it then determines to which host the selected task should be sent (location policy). In some cases, the algorithm may also have to decide on whether to accept a task transfer request from remote host (acceptance policy).

3.2 Design Objectives and Decisions

The DSF system is designed with the following objectives:

1. It should provide a convenient mechanism for the replacement of different LD algorithms so that we can treat a LD algorithm as a separate and replaceable component in our LD scheme.

2. It should be able to support concurrent running of multiple LD algorithms. This allows users to choose different LD algorithms according to their particular need.

3. It should provide major LD primitives so as to facilitate and simplify the implementation of LD algorithms. The primitives should include common functions that most LD algorithms need. For example, information about hosts in a network, load information of each LD host and the mechanism for task transfer and execution.

4. The number of LD hosts in the DSF system should be allowed to change dynamically. This provides the flexibility that more hosts can freely join the LD scheme to enhance its effectiveness. When failure of some LD hosts occurs afterwards, the LD scheme can still continue without having to terminate.

We have made the following design decisions. Firstly, the implementation of DSF should not involve kernel modification. This makes further development in porting DSF to other platform easier. Secondly, application programs should not need to modify or relink in order to take advantage of LD scheme offered by DSF. Since accessing the source code of application programs is sometimes impossible, imposing the need to modify or relink the application programs will make the number of application programs that can enjoy LD limited. Thirdly, DSF only supports LD algorithms that are dynamic, distributed and sender-initiated.

3.3 An Overview of DSF Architecture

Figure 3.1 shows the architecture of the Distributed Scheduling Framework system. The DSF in a particular host consists of three components: the DSF server, the LD library (LDlib) and the User-Agent (UA). The DSF server is an independent task that runs on every host to participate in distributed scheduling. The LDlib is the interface

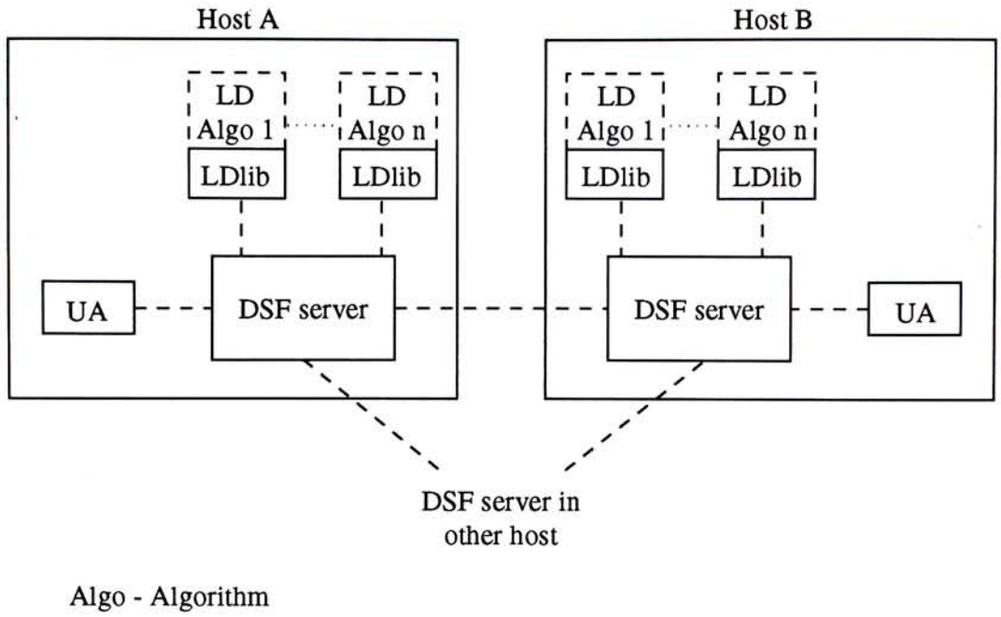

Figure 3.1: The architecture of DSF system.

that is used by LD algorithm to request different kinds of service offered by the DSF server. Each LD algorithm runs as a separate task in a host for making LD decisions. Because of this design, different LD algorithms can be running concurrently without interfering with each other. The UA is a user interface to DSF. It runs as an independent task to accept the submissions of jobs either interactively or in batch at the command level. The UA can also be used interactively to inquire status information of the local DSF server as well as global information of the whole DSF system consisting of different LD hosts. Communication among DSF server, UA and LD algorithms in the same host is basically done through Mach Interface Generator[1] (MIG) [16]. However, communication between DSF servers which are in different hosts is using the Berkeley sockets [13] interface.

3.4 The DSF server

The DSF server is a multi-threaded system server running in user space of the Mach system. It has three logically separated modules: the Load Information Module (LIM), the Movement Module (MM), and the Decision Module (DM). LIM and MM are lower level modules inside the DSF server to provide primitive services to DM which deals

[1] A program to generate code of remote procedure call for communication between a server and a client process.

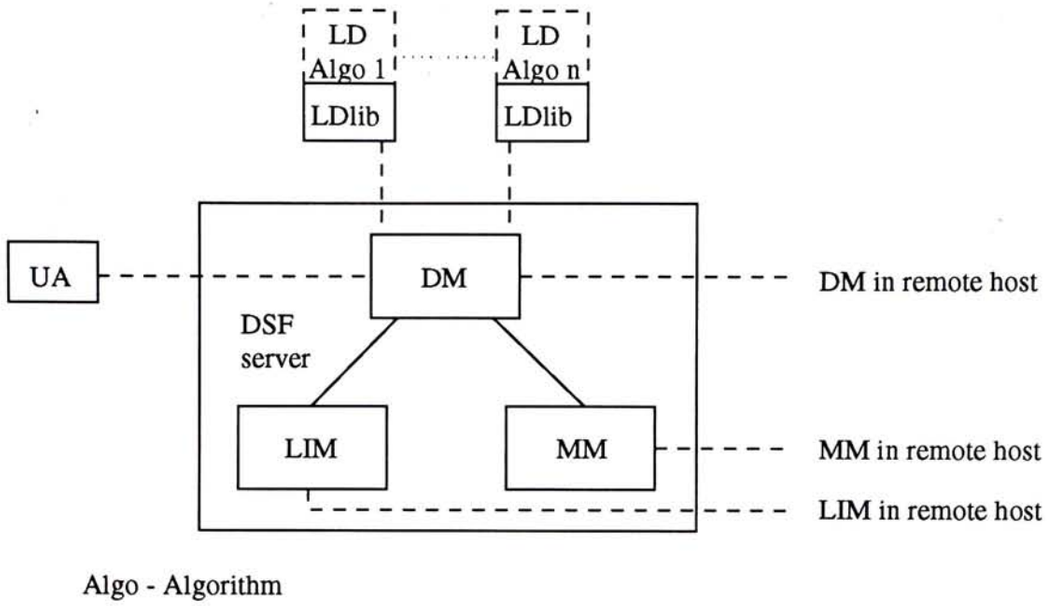

Figure 3.2: Three components inside DSF server.

with LD algorithms and User-Agent directly. This is illustrated in Figure 3.2.

The rationale behind the division of different modules inside the DSF server is to enforce the principle of separation of policy from mechanism. Since the LD algorithm is making LD decision in the policy part of LD scheme, all the interactions between the DSF server and the LD algorithm are thus collected and carried out in DM. In contrast, both LIM and MM are providing mechanism in LD scheme. Their work is in turn to support DM which acts as the agent for LD algorithms. In some implementations of LD facility [32, 33], the work of load exchange is included in LD algorithm. However, we regard the load exchange as a mechanism rather than a policy in LD scheme because DSF is decided to support multiple LD algorithms and therefore a centralized load exchange mechanism provided by LIM in the DSF server is much more appropriate than allowing each LD algorithm to have its own load exchange mechanism.

3.4.1 Load Information Module

Load Information Module (LIM) is responsible for the exchange of the current load information with each LD host and the maintenance of the status information about each DSF server. It also keeps track of all the installed LD algorithms on remote host. The organization of LIM is depicted in Figure 3.3. The Message Server, the Update Load Info unit and the Check Host unit are three permanent threads in LIM. The Host's Load Information table (host table) is the core data structure in LIM and it is shared by the

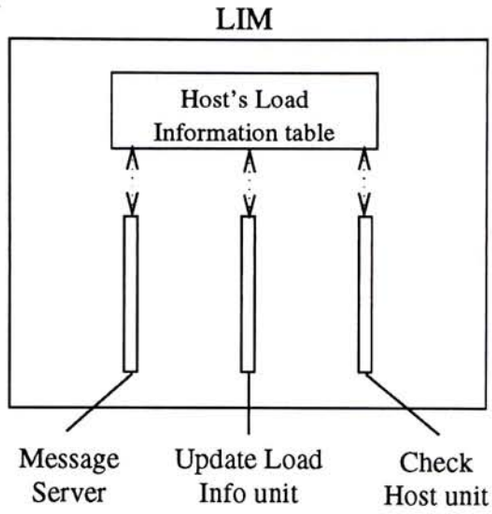

Figure 3.3: Organization of LIM.

three permanent threads.

The Message Server accepts network connection from peer LIM of other hosts. When a connection is established, it receives load information from the other end and then updates the load information corresponding to the connecting host in LIM's host table. If the host is not found in the host table, a new host entry with the load information will be added to it. Then the Message Server returns its current load information to the connecting host to complete its load exchange activity. The Update Load Info unit obtains a list of hosts participating in LD scheme from the host table, and then connects to each in turn. When a connection is successful, it exchanges its load information by first sending its load information to peer LIM and then receiving the same information. The newly obtained load information is then used to update host entry in the host table accordingly. The Check Host unit is responsible for locating any new DSF server which is not currently in LIM's host table. When the probe of new DSF server is successful, its load information will be exchanged with peer LIM and the load information of the new host is added to the host table.

The system load average[2] and the Mach factor[3] in last 5, 30 and 60 seconds are the load information provided by Mach to represent current load status in a host. They are chosen as our load indices (Section 2.1.1) for load information exchange. Since the DSF can load multiple LD algorithms, information of currently loaded LD algorithm(s) is also

[2]Load average is the average number of runnable processes divided by number of CPUs [14].

[3]Mach factor is the processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads) [14].

needed to disseminate among other LD hosts. Instead of building another mechanism for exchanging this information, we piggyback this information with load information when LIM performs load information exchange.

The LIM supports both periodic and event driven load exchange methods. In periodic mode, the Update Load Info unit connects to hosts in LIM's host table in some regular interval. In contrast, the Update Load Info unit in event driven mode only regularly samples the local load information. When the difference between the current value and the previous one is greater than some predefined value, load exchange with other LD hosts will then take place. This method has the advantage over the periodic one in avoiding useless load exchange when the fluctuation of local load is small. In order to prevent the period without load exchange from being too long, load exchange will still occur when the time difference from the previous load exchange exceeds some threshold period.

The DSF is able to allow any host to join and leave the LD service dynamically because of the cooperation between working units performing message exchange and the Check Host unit. We rely on the process of message exchange to eliminate any unavailable hosts and the Check Host unit to discover any new host running DSF server.

During message exchange between two hosts either for load information exchange between peer LIMs or for task transfer negotiation between peer MMs, a remote host is considered unavailable when the local DSF server encounters one of the following situations:

- it fails to make a connection to remote host.
- it cannot send message to remote host.
- it cannot receive message from remote host.
- the size of received message is smaller than expected.

Thus the host entry is removed from LIM's host table.

The method employed by the Check Host unit to locate new candidate hosts for LD relies on the information provided by UNIX's *rwho* service. Each host in a local network is assumed to be running a *rwhod* server. Since *rwho* service provides a list of hosts currently running in the local area network, the Check Host unit can periodically probe each host that is in this list but not in LIM's host table in turn to determine if

any new host is running the DSF server. If so, the probed host is added to LIM's host table.

In fact, the work performed by the Check Host unit is only needed to do once if no link failure ever occurs. For example, assuming there are two hosts, A and B, and the DSF server in A is started first, the Check Host unit in A connects to B for load information exchange but the DSF server in B is not started yet. Therefore LIM's host table in A will be empty. When the DSF server in B is then started, the Check Host unit in B will use the data collected by *rwho* service to probe A as A is a live host. Therefore, LIM in both A and B will recognize each other and put the other in its host table. In case either B is crashed or the DSF server in B is unavailable, LIM in A will recognize this situation when it tries to communicate with the peer in B for load information exchange. If B is available later, the DSF server in B will be restarted and hence both A and B will recognize each other again.

The work of the Check Host unit still does not need to perform regularly if a link failure occurs in the following manner. Imagine that the DSF server in A and B is already running and the other host is already in their respective LIM's host table. A link failure has occurred before B is exchanging load information with A. As a result, A is removed from LIM's host table in B. When it is time for A to communicate with B, the link problem is disappeared, B can still recognize A after the Message Server in B has served load exchange request from LIM in A. However, if the link failure still exists during the period that A is performing load exchange with B, B will be removed from LIM's host table in A. After that moment, even the link problem has gone, both A and B will not be able to recognize each other unless the work of the Check Host unit is performed again. This explains why the Check Host unit needs to perform its work regularly.

3.4.2 Movement Module

MM provides mechanism for task transfer inside the DSF server. We only support initial task placement [12] but not task migration because task migration requires substantial kernel modification [20] but it may not result in considerable performance improvement due to its high overhead [12]. Organization of MM is given in Figure 3.4. The Message Server, the Transfer unit and the Submission unit are three permanent threads. Run Job units are temporary threads and the maximum number of Run Job units allowed in MM is a system tunable parameter.

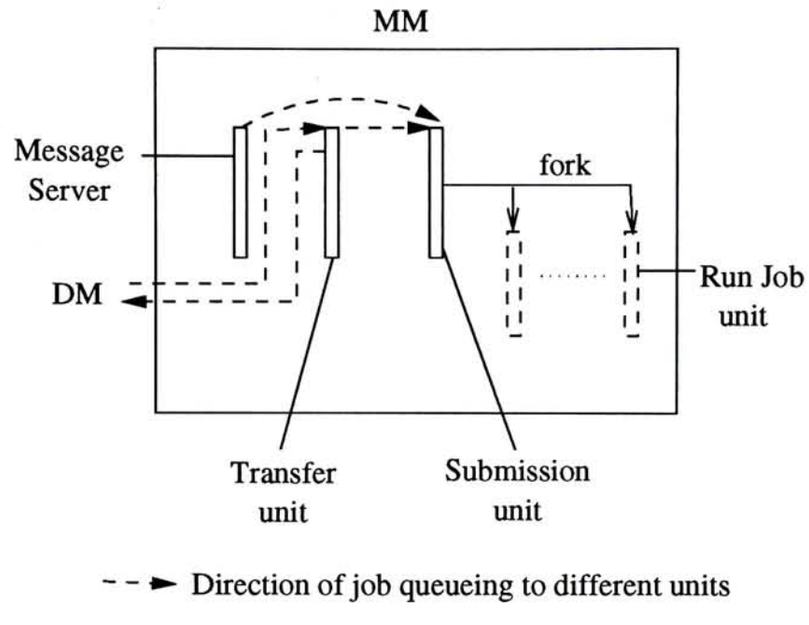

Figure 3.4: Organization of MM.

Although both LIM and MM are providing mechanism to the LD algorithm, MM is structurally different from LIM in that it does not contain a shared data structure that is accessed by its component working units. Instead, all the permanent threads are working together in order to make a submitted job get executed. We chose an asynchronous communication model in processing task transfer requests because the Message Server in MM can only serve one request at a time and allowing synchronous communication in processing task transfer will tie up the communication path between local and remote MMs until the task transfer negotiation is complete. Figure 3.5 illustrates the scenario that the communication path is tied up during task transfer negotiation when synchronous communication model is used. We divide the work performed by MM into several working units because the work performed by one unit may be required by others. For example, both the Message Server and the Transfer unit can reach a state that their currently processing job needs to be executed locally and therefore they will submit the job to the Submission unit for subsequent processing. Thus we employ the job queue model to accept job from different sources for job processing in these units. The working unit is active when there is some job waiting for processing, otherwise it remains dormant. This is achieved by using synchronization primitives from C thread package [5] (Section 4.2). The notion of job is captured by using some data structure which contains the source and target algorithm identities, the job command line, the target host for execution and others. Some of them are initially empty and filled in

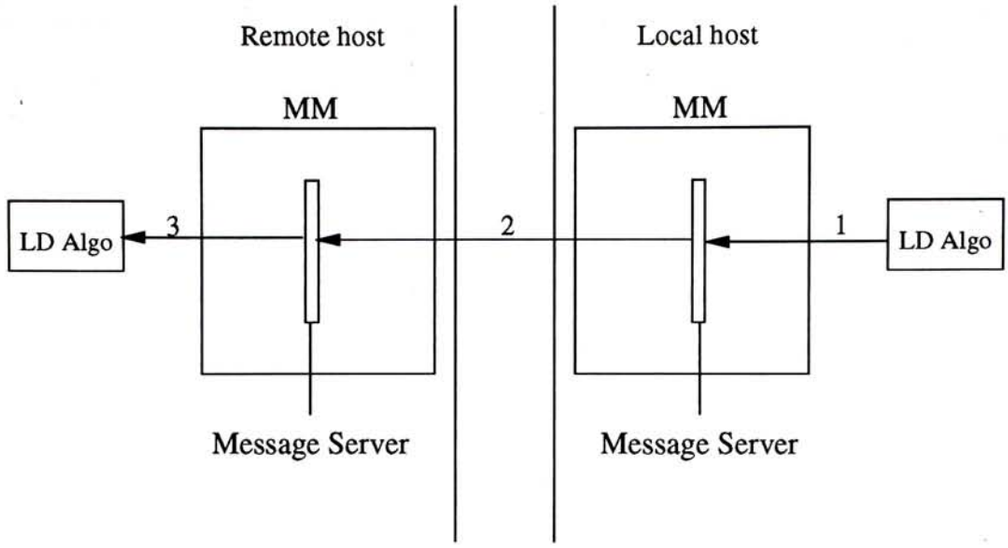

Figure 3.5: Communication path is tied up in task transfer negotiation when synchronous communication model is employed. 1) Task transfer request from source LD algorithm to local MM. 2) Task transfer negotiation between local and remote MMs. 3) Communication between remote MM and the target LD algorithm for making task transfer acceptance decision.

during task transfer negotiation.

The Message Server is responsible for serving task transfer request from peer MM. When there is a request, the Message Server determines whether the specified target LD algorithm which will make task transfer acceptance decision is loaded. If not, the algorithm with the same identity as the one which made remote assignment in originated host is chosen. If it is also not available, the task transfer request is refused. If target LD algorithm is available, the Message Server will accept or refuse the task transfer request according to previously specified decision of target LD algorithm. There are three possible decisions—ACCEPT, REFUSE and CONSULT. The first two decisions are obvious. If CONSULT is specified, the Message Server will ask target LD algorithm for acceptance decision on current task transfer request. When the final decision is ACCEPT, the transferred job information is added to the job queue of Submission unit.

The Transfer unit employs the job queue model for job processing. New job is added to job queue by DM as it directly deals with LD algorithms. In job processing, when the target location is found to be the local host, the job information is added to the Submission unit for further processing. For other target location, the Transfer unit will negotiate with MM in target location for requesting task transfer. If unsuccessful, the job will be either resubmitted to the originated LD algorithm through DM for

reconsideration or added to the Submission unit for local execution. The factor that affects the final decision depends on the number of times that a given job has been resubmitted to the originated LD algorithm. If the LD algorithm has specified the limit for re-submission, the job will be resubmitted to it until that limit has reached; otherwise the default re-submission limit will be used. During task transfer negotiation, it is possible to discover that remote host is no longer available. In this situation, the job for unavailable target location will also be either resubmitted to the originated LD algorithm for reconsideration or added to the Submission unit for local execution by using the same criteria specified before.

The Submission unit also uses the job queue model for job processing. When there is a job in its job queue, it spawns a new Run Job unit for submitting the job to local operating system for execution provided that the number of active Run Job units does not reach a predefined limit. Otherwise, the step of creating a new Run Job unit will be suspended until some old Run Job unit has exited.

Run Job unit first extracts the execution environment from the job information. Then it creates a pipe and forks a child for job execution. The child process uses the information of the extracted execution environment to setup its own execution environment and the corresponding ownership information. Then it directs its standard output and error to the Run Job unit (in parent) through previously created pipe. The Run Job unit then establishes a connection with the originated User-Agent for the following purposes:

- to indicate that UA's previously submitted job is started now
- to forward any output from executing job to the UA
- to indicate the job execution is terminated

Figure 3.6 illustrates the communication among the UA, the Run Job unit and the executing job command. With the above design, we are able to direct standard output and error from remotely executing job to the UA in originated host. However, since our design is following the same model as UNIX *rexec* service, we also suffer the same limitation that we cannot allow the transferred task to be an interactive application.

3.4.3 Decision Module

DM provides registration service for LD algorithms that are loaded in the local host. It maintains information about the status of locally registered algorithms by

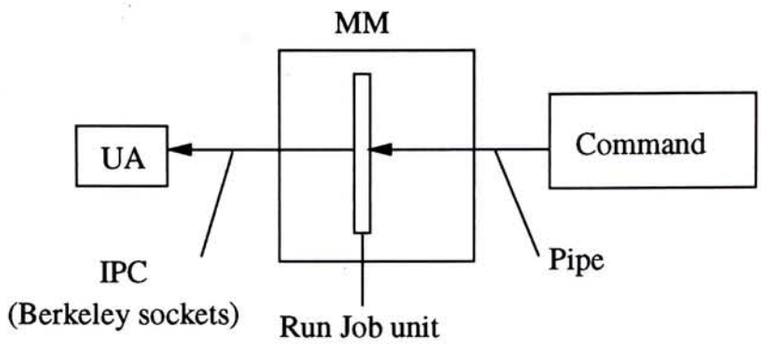

Figure 3.6: Communication among the User-Agent, the Run Job unit and the executing job command during job execution.

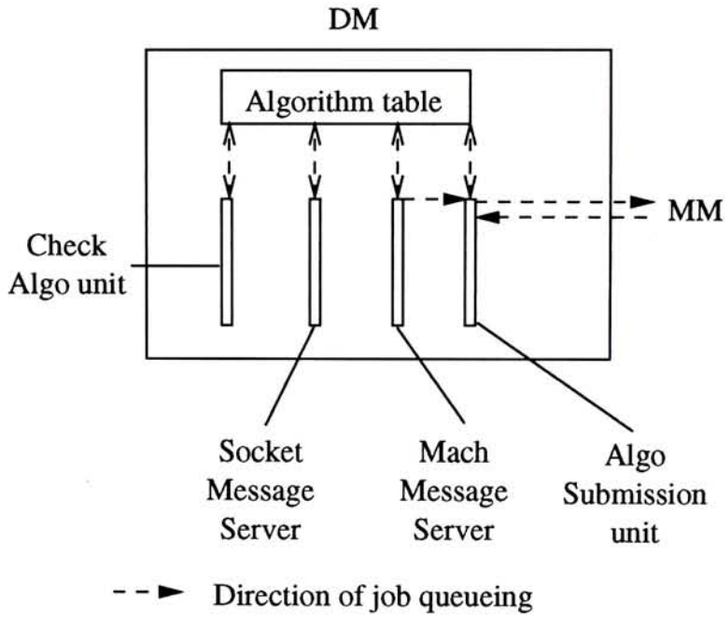

Figure 3.7: Organization of DM.

regularly probing each of them. It also provides information to queries about DSF server maintained information such as host status, load information and algorithm information to both LD algorithms and UA. Figure 3.7 shows the organization of DM. DM consists of the Mach Message Server, the Socket Message Server, the Check Algo unit and the Algo Submission unit. The Algorithm table is the core data structure in DM and it is shared by the above four threads.

The Mach Message Server provides the following services:

- it accepts LD algorithm registration and updates DM's Algorithm table accordingly.
- it accepts job submitted by UA and then adds it to Algo Submission unit.

- it provides the load information of all LD hosts and the information of loaded LD algorithms in each host to either LD algorithm or UA.

The role of the Socket Message Server is to support direct send-and-receive communication between two LD algorithms. The Socket Message Server accepts connection from two sources. The first is from locally loaded LD algorithm and the second is from peer DM. When the connection is from LD algorithm, it checks whether the specified algorithm in target host is loaded from information obtained from LIM's host table. If the algorithm is not available, an error is reported to originated LD algorithm. Otherwise it connects to DM in target host for sending a message for the originated algorithm. When the connection is from peer DM, it checks whether the specified algorithm is loaded from DM's Algorithm table. If not, an error is returned to connecting DM. Otherwise, the message received from peer DM will be forwarded to the specified algorithm. The reply, if any, from the specified algorithm is then sent to the originated algorithm along the same communication path but in opposite direction.

The Algo Submission unit employs the job queue model for job processing. It processes job from UA (first time job submission) or MM (job re-submission) by making Mach RPC to the corresponding LD algorithm. When it fails to submit the job to the corresponding algorithm, it will send the job to MM for local execution.

The Check Algo unit regularly obtains a list of loaded LD algorithms from the Algorithm table and probes each in turn. If the probing is unsuccessful, the corresponding algorithm is removed from the table. Since a LD algorithm exists as a separate task and its association with DM is only through Mach IPC mechanism, it may be dead or killed anytime without notice. Therefore, the function of the Check Algo unit is to monitor the liveness of each LD algorithm.

DM maintains the Algorithm table of all locally registered LD algorithms with the following information:

- algorithm identity and description
- default decision (ACCEPT, REFUSE and CONSULT) to remote task transfer request
- limit for job re-submission to algorithm for reconsideration
- send right [14] to each algorithm port which is needed for Mach Message Server to talk to the algorithm

- a port number for Socket Message Server to communicate with an algorithm if the algorithm intends to have direct communication with its peer algorithm.

This information is filled in during algorithm registration. As each LD algorithm is implemented as a separate task, it must register itself with DM so that information can be directed to the LD algorithm later. Once a LD algorithm is registered with DM, we call it loaded into DSF since it is now visible to the system, and it can be specified by user during job submission with UA.

DM provides a default algorithm which always accepts all submitted tasks and sends them to MM for local execution. With this default algorithm, we can guarantee that a task will always have a host to run even if there is no user defined LD algorithm available.

DM is responsible for answering queries from LD algorithms through LD library interface and User-Agent. When information about loaded algorithm for a host H is needed, DM obtains the information from LIM if H is not a local host. Otherwise, DM directly obtains the information from the Algorithm table. Then the information is returned to either UA or LD algorithm. DM does not keep the information of loaded algorithms in other hosts because the list of hosts which run a DSF server changes dynamically and inconsistence will arise if DM maintains a separate list.

3.5 LD library

DSF is designed to facilitate the implementation of LD algorithms. However, it is still the algorithm designer's responsibility to provide the core functions in algorithm task since each algorithm is different in terms of its load distribution policies and information used to derive those policies. DSF provides a set of commonly required services that LD algorithms can use so that the programming of a LD algorithm can be simplified.

The following service functions are provided by LD library:

- To obtain a list of hosts with their current load information in LD scheme.
- To obtain a list of loaded LD algorithms for a particular host.
- To make task transfer request for either local or remote execution.
- To communicate with a particular LD algorithm in remote host by using send-and-receive communication model with fixed message size.

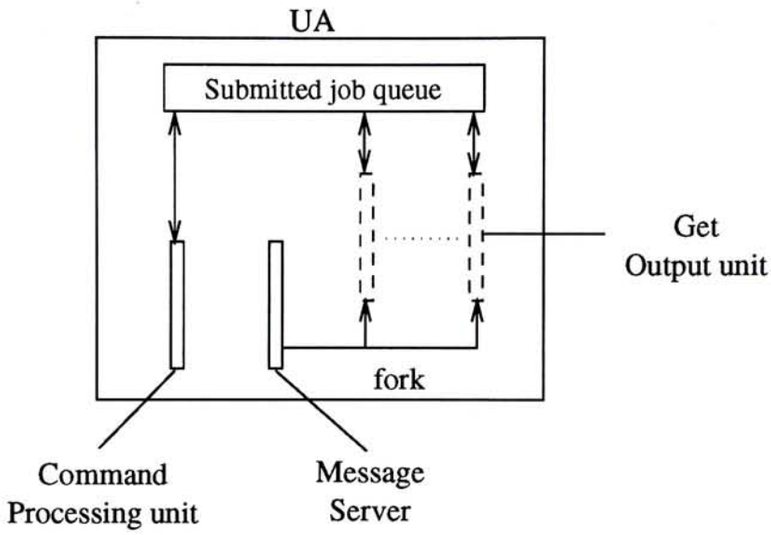

Figure 3.8: Structure of UA.

The detailed description of the above service functions can be found in Appendix A.

Since the procedure for LD algorithm registration is standardized, we are able to provide a stub function for the algorithm registration so that the algorithm designer can be free from writing the same procedure in each algorithm repeatedly. As a result, the algorithm designer only needs to specify the information essential for algorithm registration without the need to invoke any function. The resulted LD algorithm will be automatically registered when it is compiled with the stub function. The sample implementation of some LD algorithms by using our LD library is given in Appendix B.

In algorithm registration, the most important issue is the algorithm identity which is used to uniquely represent a particular type of algorithm in a DSF system. In order to avoid the situation when the same algorithm acquires different algorithm identity when it registers in different hosts, we assume some kind of mechanism exists to register the type of services an algorithm provides. When a new algorithm is written, it is the writer responsibility to acquire the identity beforehand and use the same algorithm identity and description on all hosts.

3.6 User-Agent

The User-Agent (UA) is a user interface to the local DSF server for job submission and query for loaded algorithm information so that users can make a choice on source and target LD algorithms. The structure of UA is given in Figure 3.8. UA consists of two permanent threads: the Command Processing unit and the Message Server. Get

Output units are temporary threads dynamically created by the Message Server. The data structure shared by the Command Processing unit and all Get Output units is a queue of submitted jobs which are accepted by UA either interactively or through its command line interface.

We decided to keep the queue of submitted jobs in UA rather than in the local DSF server because of the following reasons. Firstly, we can save making a substantial number of IPC to local DSF server when many users are inquiring their submitted job status information. Secondly, only one communication channel is needed instead of two for each job executed by the Run Job unit in MM. Two communication channels are needed by the Run Job unit because one is for directing job's standard output and error to the UA in originated host and the other is for indicating the current job status to the DSF server in originated host since the DSF server keeps the submitted job queue. However, this also creates the problem that the DSF server in the originated host has no way to detect if there is any error occurring in job's output channel. If job's output is first directed to the DSF server in originated host and then it is relayed to UA (two communication channels are still being used), the DSF server will be able to monitor the output channel status but the extra work in relaying job's output to UA is required. In contrast, putting the submitted job queue in UA makes directing job's output straightforward as only one communication channel is required between Run Job unit and UA. Monitoring job's status can also be achieved through receiving control messages from the communication channel. Thirdly, we anticipate that the DSF server in originated host will become a bottleneck in relaying job's output to UA if the submitted job queue is kept in it. It is because a separate thread will be needed to handle relaying one job's output and if the system has a lot of jobs generated, the number of threads consumed by the DSF server will be probably exhausted.

The Command Processing unit is responsible for processing all kinds of commands submitted to UA either interactively or at the command line. When a user issues a command, the unit does the input error checking to ensure that a valid command is entered, and the correct number and format of arguments are given if the issued command requires them. The following are the major functions served by the Command Processing unit when UA is in interactive mode:

- To provide the current load information of all hosts in LD scheme. For each host, the algorithm identity and description of all loaded algorithms are displayed. The information is obtained by making a Mach RPC to DM in local DSF server.

- To accept job submission for one job or using a job file for batch jobs. No matter what format is used, each job entry, either input by a user or specified in a job file, should include at least a command for execution, and optionally specify both source and target algorithm identities. If none is given or only the target algorithm is specified, the default algorithm will be used as the source LD algorithm and the job will be executed in the local host. When the target algorithm is missing, it will be defaulted to the same identity as the source algorithm.

  When the unit has accepted a job submission, it creates a job structure which contains the current execution environment such as current working directory, user identity and etc. Then it verifies whether the specified source algorithm is currently loaded by making a Mach RPC to DM. If not, an error message is returned to the user. Otherwise, it makes a Mach RPC to DM for actual carrying out the job submission process. Before submitting the job to DM, a unique job identity is acquired and the corresponding information about the submitting job is also queued up in the queue of submitted jobs.

- To display the redirected standard output and error of currently executing job. We do not allow the output to display to the screen directly because it will cause a mess as the screen is also used by interactive command interface of UA. Therefore, all the output from previously submitted jobs are stored in a temporary file and it is only displayed when a user requests it. Since all the output are kept in one file, this creates an inconvenience that output from different jobs could be intermixed and examining the output corresponding to each job becomes difficult. Continuously monitoring the output of a particular job is also difficult because the user needs to repeatedly issue the corresponding command. Thus we provide an alternative interface for viewing job's output if the user is working in the X Window System. With such window system, the user can optionally choose to have the output of each running job directed to an independently created window. This solves the previously mentioned two problems occurring in text mode terminal.

- To indicate the current status of all submitted and non-finished jobs. The job status can be in one of two states: SUBMITTED or RUNNING. SUBMITTED indicates that the job is accepted by the local DSF server but the target host has not started to execute it. It is possible that the job is inside LD algorithm and waiting for target location decision, in the state of task transfer negotiation or in

the queue of Submission unit in DSF server and waiting for a free Run Job unit. RUNNING shows that a target host is now executing the job and the running host is also displayed. Since a unique job identity is associated with each submitted job, two jobs with the identical command can still be differentiated.

As the Message Server in UA is a concurrent server [25], it creates a new Get Output unit for serving each connection from the Run Job unit in MM. The Get Output unit receives the following three types of message from the Run Job unit:

- Start message
- Termination message
- Output message

Start message indicates that the Run Job unit has already started executing the job and it is now waiting for job's output. Since the message also includes the job identity which was generated when the job information is put into the queue of submitted jobs, the identity can now be used to update the job status that it is in RUNNING state. When a user chooses to display the job's output in an X client window, the window is also created at this stage. Termination message indicates that the Run Job unit has found that the job has finished execution. Therefore the Get Output unit cleans up the information of finished job from the queue of submitted jobs and then reports the completion of that job to the user screen. Output message indicates that this message contains a certain amount of output from the running job. The Get Output unit then directs them either to the temporary file for storing jobs' output or the previous created X client window depending on the type of output that the user has previously chosen. When Get Output unit detects any connection failure, it will perform the similar job as of receiving Termination message but report to the user that the job is abnormally terminated.

Chapter 4

The System Implementation

Since Mach supports multiple threads of control within a single address space of a Mach task, we exploit this feature heavily in the implementation of the DSF. This also makes our implementation very different from other LD facilities [7, 33, 27, 32] which were implemented in a traditional UNIX system without support of multithreaded programming interface. In this chapter, we focus on the implementation issues of DSF under Mach programming environment.

4.1 Shared data structure

In a multithreaded programming environment, it is natural that data are shared among related threads. However, allowing access to shared data by each thread concurrently can result in data inconsistency or race condition[1]. The C threads package [5] in Mach provides a mutual exclusion primitive called mutexes. When a mutex is locked, any attempt to lock it again will block until it is released. Thus it can be used to allow only one thread at a time to access a shared data. The usual method of accessing a shared data by any thread involves the following steps:

1. lock a mutex variable for the shared data
2. manipulate the shared data
3. unlock the mutex variable

[1] A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place [24].

However, a deadlock problem can occur when a thread has attempted to lock a mutex variable that it has already held. This can happen when a thread is accessing the same shared data that it has previously locked from another path indirectly [16].

Although we can use mutex for accessing shared data, associating each shared data with a mutex variable and accessing several shared data at the same time in order to accomplish a work will make programming prone to error because a lot of locking and unlocking are needed to perform. It may possibly make the problem of deadlock occur more easily. In order to avoid these problems, we group data of similar nature in one data structure, or we call it an *object*, and associate it with one mutex variable. As a result, we lock the shared data at an object level rather than at each data item level. The object concept also affects our system design stage. For example, when designing what should be performed in different modules inside DSF server, we always try to identify an object that can be shared by different working units in a module. As a result, Host's Load Information table and Algorithm table were identified as the object in LIM and DM respectively.

Another aspect that needs to consider in multithreaded programming is the level of granularity that we use to protect shared data with mutexes. For instance, if we have a linked list that needs to be shared between two threads, we may choose a coarser granularity by using a mutex variable to lock the whole list. That makes two threads concurrently read the linked list impossible and we may lose the possible benefit of parallelism when the two threads are accessing different members of the list. On the contrary, if a mutex variable is associated with each member, the two threads can access the list concurrently. However, the overhead in locking and unlocking the mutex variable of each member in order to locate the required one may be so large that it offsets the benefit brought by the parallelism from fine granularity. We choose the coarse level of granularity to protect the shared data object in DSF because it is much straightforward to implement and have lower possibility of deadlock. In order to avoid the problem that the shared data object is locked for too long, our code of accessing shared object is written in such a way to minimize the period of locking shared object.

Since our shared data structures, or objects, are protected by mutexes, accessing them must be done through locking the corresponding mutex variables. We may be tempted to do this kind of locking anywhere in our code to access the necessary shared data object. As a result, the code for locking shared data object is scattered in the programming logic of different threads. The possible problems of locking the shared

data object in this way include:

- We have a lot of redundant code just for locking and unlocking the shared data object.
- The logic of locking/unlocking shared data object is intermixed with programming logic of each thread and this decreases the program readability.

We avoid the above problems by following the principle that shared data object should not be accessed by other threads directly. Instead, we provide a set of functions decided explicitly for accessing a particular shared data object and threads should access the shared data object indirectly using the provided functions.

We use the implementation of the Host's Load Information table which is the shared data object in LIM to illustrate the above principles. The table is defined as the following object:

```
/* host info object but excluding local host info */
struct {
  int no; /* no. of other hosts participating in LD scheme */
  struct host_ent head; /* head of a list of host entry */
  struct mutex lock;
} hosts_obj;
```

Both `hosts_obj.no` and `hosts_obj.head` are shared data. We use a mutex variable `hosts_obj.lock` to protect all the data inside `hosts_obj` instead of associating a mutex variable with `hosts_obj.no` and adding a mutex variable field in `struct host_ent`. The following are the functions that can directly access the shared object `hosts_obj`:

```
/* function prototypes */
void hosts_init(void);
int addhost(struct host_ent *);
int updhost(struct host_ent *);
int gethostinfo(int, struct h_info **);
int gethostalgo(const char *, struct algo_info **, int *);
void delhost(struct in_addr);
```

`hosts_init()` does the initialization on `hosts_obj` and the remaining functions are to manipulate the host list in `hosts_obj`.

An initialization function is always needed for each shared data object because the mutex variable needs to be initialized before it can be used [16]. In initialization function `hosts_init()` below

```
/* initialize hosts_obj object */
void
hosts_init(void)
{
  hosts_obj.no = 0;
  hosts_obj.head.next = NULL;
  mutex_init(&hosts_obj.lock);
}
```

besides using `mutex_init()` to initialize the mutex variable `hosts_obj.lock`, we also initialize the value for the two fields in `hosts_obj`.

Among those shared data object manipulation functions, we use `addhost()` to illustrate how we access the shared data object

```
/*
 * add a new host entry to host_obj
 *
 * return: 0 - successful
 *        -1 - host is present
 */
int
addhost(struct host_ent *host)
{
  struct host_ent *pt1;

  host->next = NULL;
  mutex_lock(&hosts_obj.lock);
  for (pt1 = &hosts_obj.head; pt1->next != NULL;) {
    pt1 = pt1->next;

    if (memcmp(&pt1->hostid, &host->hostid, sizeof(pt1->hostid))
        == 0) { /* duplicated host */
      mutex_unlock(&hosts_obj.lock);
      return -1;
    }
  }
  pt1->next = host;
  hosts_obj.no++;
  mutex_unlock(&hosts_obj.lock);
  return 0;
}
```

Since the statement

```
host->next = NULL;
```

does not need to access the shared data object, we put it outside the region that is protected by mutex variable `hosts_obj.lock`. In the *if* statement inside the *for* loop, once we determine that adding the host to the shared data object is impossible because the host is already there, we immediately unlock the mutex variable so that other thread can gain access to it.

In the implementation of functions that have to directly access the shared data object, we have the following observations:

- The code inside the function is mostly protected by mutex variable since its purpose is to access the shared object.
- If there is any code that does not need to lock the shared object, it should be done outside the region protected by mutex variable.
- The code should determine if there exists any condition that makes further execution of the function useless. If so, unlock and terminate the function so that other threads can access the shared object earlier.

4.2 Synchronization

Besides the mutual exclusion primitive, C thread package also provides a synchronization primitive, called condition variables. Condition variables are used when a thread needs to wait for some condition to be true before it can proceed and such condition is set by another thread. In practice, a shared data object is involved in synchronization process. When a thread has locked a mutex variable for a shared object, it examines whether the shared data object has been in a state that it is waiting for. If not, it indicates that it is waiting for such condition by executing `condition_wait()` on a condition variable. After the execution, the thread is blocked and its previously locked mutex variable is also released. Another thread which performs a different function than the blocked one can then lock the shared data object. If it has found that the state of the shared data object after its modification is what some other thread is waiting for, it executes `condition_signal()` to wake up the suspending thread before unlocking the shared data object. After the suspended thread is unblocked, it rechecks whether the condition that it has previously waited for is true. If so, it proceeds normally; otherwise it `condition_wait()` the condition again.

The synchronization primitive is essential to the implementation for the job queue which is used in Transfer and Submission units inside MM. It is also used for control the maximum number of Run Job units allowed in MM. The following shows the definition of job queue in Submission unit

```
/* job queue in submission unit */
struct que_obj {
  int no; /* no. of job entry */
  struct job_ent *head; /* list of jobs */
  struct mutex lock;
  struct condition non_empty; /* condition variable */
} subm_q;
```

The following functions are accompanied with the job queue (object) for accessing the shared object directly

```
/* function prototypes */
void subm_q_init(void);
void add_subm_job(struct job_ent *);
struct job_ent *deq_subm_job(void);
```

`subm_q_init()` is an initialization function. `add_subm_job()` is used by Transfer unit and Message Server in MM to add job to the job queue. `deq_subm_job()` is used by Submission unit itself to dequeue job for subsequent processing.

In `deq_subm_job()`

```
/*
 * delete job in submission queue
 *
 * return: dequeued job
 */
struct job_ent *
deq_subm_job(void)
{
  struct job_ent *job;

  mutex_lock(&subm_q.lock);
  while (subm_q.head == NULL)
    condition_wait(&subm_q.non_empty, &subm_q.lock);

  job = subm_q.head;
  subm_q.head = subm_q.head->next;
  subm_q.no--;
  mutex_unlock(&subm_q.lock);
  return job;
}
```

when `deq_subm_job()` is called, it locks the mutex variable `subm_q.lock` and then determines if the job queue is empty. If so, it calls `condition_wait()` with condition variable `subm_q.non_emtpy`. The calling thread will be blocked until it is waken up by another thread which has called `condition_signal()`.

In `add_subm_job()`

```
/* add job to submission queue */
void
add_subm_job(struct job_ent *job)
{
  struct job_ent *pt;

  job->next = NULL;
  mutex_lock(&subm_q.lock);
  if (subm_q.head == NULL) {
    subm_q.head = job;
    subm_q.no = 1;
    condition_signal(&subm_q.non_empty);
    mutex_unlock(&subm_q.lock);
    return;
  }

  /* advance to the end of the queue */
  for (pt = subm_q.head; pt->next != NULL; pt = pt->next)
    ;

  pt->next = job;
  subm_q.no++;
  mutex_unlock(&subm_q.lock);
}
```

after the mutex variable `subm_q.lock` is locked, the *if* statement adds the job to job queue `subm_q` if it is previously empty. Since non-empty is the condition that a thread running `deq_subm_job()` possibly waits for, therefore it `condition_signal()` the condition variable `subm_q.non_empty` to wake up the Submission unit if it is previously blocked by calling `deq_subm_job()`.

4.3 Reentrant library

Although Mach supports multithreaded programming by providing the C thread package, the degree of support is only barely sufficient because most of the functions provided by the C library are non-reentrant or thread-unsafe. Reentrant functions are

those that allow two or more threads to call them simultaneously without causing any indeterminate result [4]. The problem of non-reentrant functions is that they use either global or static data and thus make them become "shared data" when the functions are called by different threads concurrently. A typical example of non-reentrant function is `strtok()` which is called in succession to obtain a token from a string. As a result, precautions are needed in using common C library functions so that we are not directly using those non-reentrant functions.

As far as making system call is concerned, most of the time we do not encounter the non-reentrant problem. However, the system call usually uses the global variable `errno` to indicate the type of error that it has encountered when calling it is unsuccessful. This may cause the problem in determining the real cause of system call failure because when a thread is going to examine `errno` for determining the cause of system call failure, `errno` may be modified by another thread which has encountered another system call failure. We do not have a clean solution to solve this problem but suggest a way to write the code for examining `errno` so that the chance of encountering the above problem can be reduced. For instance, when we use the system call `write()`, instead of writing in the usual manner

```
n = write(...);
if (n < 0) {
  ...
  report error with errno
  ...
}
```

we would write in the following way

```
n = write(...);
if (n < 0) {
  int err = errno;

  ...
  report error with err
  ...
}
```

In this way, `err` will have the same value as that in `errno` when `write()` error occurs provided that `errno` is not modified for the period from the time `errno` contains the true error code to the time that `errno` is assigned to `err`.

We now illustrate how we call a non-reentrant library function safely. The Run Job unit in MM is responsible for executing job. Before job execution, it needs to obtain the information from a password file of local system for a particular user identity (UID) so that it can set up the execution environment for the job. The library function for obtaining such information is

```
struct passwd *getpwuid(int uid);
```

Since `getpwuid()` returns a pointer to some structure, it must be using a static data structure and therefore non-reentrant. We lock the function before calling it

```
struct mutex fn_lock[MAX_FN_LOCK];
struct passwd *tmppwd;
struct {
  ...
} pwd; /* modified from struct passwd */

...

mutex_lock(&fn_lock[GETPWUID]);
if ((tmppwd = getpwuid(job->job_env.uid)) == NULL) {
  mutex_unlock(&fn_lock[GETPWUID]);
  ...
}

/* copy selected fields from tmppwd to pwd */
...
mutex_unlock(&fn_lock[GETPWUID]);
```

In order to make the locking period as short as possible, we use another variable `pwd` to store the information pointed by `tmppwd`. Once this is done, the function can be unlocked and we can use the data returned by `getpwuid()` safely. If we do not use the variable for holding the returned value, the function locking period will be unnecessarily longer.

Adding a function lock for each non-reentrant function is not always possible to solve the non-reentrant problem because two different functions can further call the same function later during its invocation. For such type of functions, we can only identify them by examining the source code of the library. Otherwise, we can encounter error during the invocation of those functions.

In the implementation of load exchange mechanism in LIM, we encountered the program error which is resulted from invoking two different but similar functions. When

Check Host unit in LIM locates a new LD host, it calls `gethostbyname()` to obtain some address structure for making subsequent network connection. When Message Server in LIM has received a connection from other peer, it calls `gethostbyaddr()` to determine the hostname of the peer. Locking each function separately before calling cannot solve the program error problem. However, when using the same function lock for calling either `gethostbyname()` or `gethostbyaddr()`, the program error does not occur again.

4.4 Interprocess communication (IPC)

4.4.1 Mach IPC

Mach provides interprocess communication among threads via structures called *ports*. Ports are protected objects and only Mach tasks with appropriate send or receive rights can access a port [3]. Mach IPC can be transparently extended over a network by using a network message server [11]. This results in making Mach port location independent which is a very distinguished feature of Mach IPC.

Using Mach IPC directly in program is usually difficult because of its complex syntax and the tedious work in handling its *typed* message [15]. Moreover, Mach IPC is mostly used in the program following a client-server communication model which involves the steps of packing, sending, receiving and unpacking message. Since this procedure is routine, Mach provides a program called Mach Interface Generator (MIG) for generating remote procedure call (RPC) for communication between a server and a client, and therefore relieving the burden of programmer in writing those routine procedures.

Mach IPC is used in communication between DM and LD algorithms. MIG is used to generate a set of RPCs to provide services to LD algorithms. In particular, since DM needs to communicate with LD algorithms after they have registered, a Mach port information must be supplied during algorithm registration. Although the type of Mach port is defined as `mach_port_t` which in turn is an integer type, we cannot pass the number of Mach port to DM directly because Mach port is an object with send or receive rights associated with it. Getting only the Mach port number does not mean that a Mach task (DSF server) has acquired the right to send to or receive from that port (in LD algorithm). When writing the MIG specification for DM in providing algorithm registration service

```
routine algo_reg
```

```
(
        server:         mach_port_t;
        info:           ptr_t;
        algo_port:      mach_port_make_send_t;
  out   ret:            int
);
```

we need to inform MIG that in making this RPC, `algo_port` is a Mach port which the sender (LD algorithm) intends to give the send right to the receiver (DSF server) by specifying it as a type of `mach_port_make_send_t` instead of just a plain `mach_port_t`. In fact, the Mach port number acquired in DSF server can be different from the one in LD algorithm. This shows that Mach port is really an object rather than a number.

4.4.2 Socket IPC

In early implementation of DSF, all IPCs including those across a network were done through Mach IPC. It is because the location transparent feature of Mach port makes the implementation very convenient. For example, a Mach port is created in UA and its send right is passed to the DSF server, which may be in remote host, so that MM in the server can send job's output to the UA by using the send right of the port without concerning where the UA is. However, when we used DSF for conducting experiments for studying LD algorithms, we discovered that the network message server, which is responsible for making Mach port location transparent, is extremely unstable under high system load and may cause system panics. As a result, we chose Berkeley socket [13] as the replacement for any network IPC and restricted Mach IPC for communication only in the same host.

When MIG was used for generating RPC code for communication among DSF servers, the corresponding Message Server can process only one request at a time or it is an iterative server [25]. Therefore, we also replace the old Message Server by an iterative Message Server using Socket IPC. This kind of Message Server is found in LIM and MM. The following is the Message Server in LIM

```
void
msg_server(void)
{
  int sd; /* socket descriptor */
  int err;

  sd = socket(...);
```

```
  if (sd < 0) {
    err = errno;
    report error with err
  }
  ...
  if (bind(...) < 0) {
    err = errno;
    report error with err
  }

  listen(...);

  for (;;) {
    int con_sd; /* connected socket descriptor */

    con_sd = accept(...);
    if (con_sd < 0) {
      err = errno;
      report error with err
    }

    /* receive, process and reply message */
    ...
  }
}
```

Implementing an iterative server in multithreaded environment is similar to doing the same in traditional UNIX environment because the system calls involved are thread safe. If not, some precautions are needed to avoid the typical non-reentrant problem.

When the server needs to consecutively read a message and perform work based on received message for several times, we optimize this type of operation by aggregating them into one read operation with message size equal to the sum of the original smaller messages and delaying the original message processing work until the aggregated message is received. We perform this operation as follows

```
int read_cnt, err, n;
char buf[SIZE];

...
read_cnt = 0;
err = 0;
while ((n = read(con_sd, buf + read_cnt, SIZE - read_cnt)) != 0) {
  if (n < 0) {
    err = errno;
```

```
    ...
  }
  read_cnt += n;
}
if (err != 0) {
  /* report error */
  ...
}

/* process buf */
...
```

This can be done because in client-server communication model, the server works by repeating the steps of receiving, processing and replying message and the client works by sending request, receiving reply and processing. Thus we can implement the client in the following way. When the client has sent all its messages, it can shutdown the socket in sending direction so that `read()` in server side will return 0. However, there are other error conditions also giving this result so we check `err` to ensure that the return value of 0 is resulted from socket shutdown from the client. Once the aggregated read is finished, we can process the received buffer `buf`.

Since iterative servers are less efficient when they need to handle a lot of requests from client, we therefore implement a concurrent message server in DM (Socket Message Server) and UA to enhance its performance. The following is the Socket Message Server in DM

```
struct mutex var_lock[MAX_VAR_LOCK];
...

/*
 * this is a concurrent message server
 */
void *
dm_msg_server(void *unused)
{
  int sd; /* socket descriptor */

  ...
  sd = socket(...);
  if (sd < 0) {
    /* report error */
    ...
  }
```

```
  if (bind(...) < 0) {
    /* report error */
    ...
  }

  listen(...);

  for (;;) {
    int con_sd; /* connected socket descriptor */

    mutex_lock(&var_lock[CON_SD]);
    con_sd = accept(...);
    if (con_sd < 0) {
      /* report error */
      ...
    }

    /* fork a new thread to handle the request */
    cthread_detach(cthread_fork(proc_dm_msg, &con_sd));
  }
}

/*
 * process request to DM socket message server
 */
void *
proc_dm_msg(void *con_sd)
{
  int sd;
  ...

  memcpy(&sd, con_sd, sizeof(sd));
  mutex_unlock(&var_lock[CON_SD]);
  ...
}
```

The implementation of the concurrent server is similar to the iterative server except that we spawn a new thread to handle each request. However, the variable `con_sd` needs to be handled differently as it is shared between the Socket Message Server `dm_msg_server` and its working unit `proc_dm_msg()`. We use the mutex variable `var_lock[CON_SD]` to achieve the synchronization between Socket Message Server and its working unit. This is required because we have to ensure that the connected socket descriptor `con_sd` is not modified by another invocation of `accept()` until the thread `proc_dm_msg()` has safely copied its value to its local variable `sd`.

Chapter 5

Experimental Studies

In previous chapters, we emphasized that DSF is a tool that allows users to submit their job for execution with the benefit of automatic load distribution. In fact, DSF can also be used to conduct experiments for studying LD algorithms purpose. In this chapter, we carry out several experiments on studying some well-known LD algorithms. The objective of this experimental study is not to perform a comprehensive study on the chosen algorithms but to demonstrate that our DSF can actually be used for experimental study of LD algorithms.

5.1 Load Distribution algorithms

In a LD algorithm, it consists of transfer policy, location policy and optional acceptance policy. The transfer policy determines whether to process a task locally or remotely and the location policy determines to which node a task selected for transfer should be sent. The acceptance policy plays its part when the algorithm is considering task transfer request from remote host. The following three sender-initiated LD algorithms from [8] were implemented for our algorithm study. They are all having the same transfer policy but different location and acceptance policies.

The transfer policy adopted by the three algorithms is threshold policy. When a task is submitted by Algo Submission unit in DM to a particular algorithm, the algorithm fetches local load information from DSF server by using LD library interface. Then it compares the current load with some threshold value predefined when the algorithm was compiled. If the load is above the threshold value, it will invoke its location policy; otherwise it will choose the local host as target location and send the task information

to DSF server for local execution.

The location and acceptance policies employed by the three LD algorithms are explained below:

RANDOM When a task is decided for task transfer to a remote host, a host is selected randomly from a list of LD hosts as the target destination. However, if the local host is the only host in LD scheme, local host will still be selected for task execution. This algorithm always accepts remote task transfer request.

THRHLD For the threshold location policy, the algorithm differentiates whether a task is submitted first time or more than one time to it. In the case of the first time, it randomly selects a host from a list of LD hosts as the target destination. When the task was submitted to it previously, this shows that it had made a bad decision before and the algorithm will try to select another LD host as destination. For the acceptance policy, it will accept a remote task transfer if it finds its current load is below the threshold value specified in the transfer policy.

LOWEST The algorithm determines a host with the lowest load among the LD hosts. If the lowest load is greater than or equal to threshold value used in transfer policy, local host is chosen. Otherwise, the previously chosen host is selected as the target destination. This algorithm also always accepts remote task transfer request.

We have an algorithm called MIXED. Strictly speaking, this is not a true algorithm but a condition of having the above three algorithms running concurrently with each having the same probability of being chosen by the artificial load generator (see Section 5.2). Since our DSF allows multiple LD algorithms to be loaded at the same time, we are interested in seeing how this ability can improve the system performance with respect to individual LD algorithm.

For comparison purpose, we also have two pseudo LD algorithms. The first is called NoDSF which has no DSF server running in each host during the experiment. The second is called DEFAULT which has DSF server running in each LD host but the source LD algorithm was explicitly chosen as DEFAULT. Consequently, all tasks submitted to DEFAULT will be executed locally with the overhead of using DSF.

5.2 Experimental environment

We have used three 486 PCs for conducting the following experiments. Each PC is having a 486DX2 66MHz processor, 16MB main memory and is running CMU Mach 3 operating system with microkernel MK83 and BSD UNIX server UX42. The PCs are connected to a local area network.

We have implemented an artificial task that takes a parameter on how much CPU time it needs to consume. The task runs in an infinite loop and keeps track on the amount of CPU time that it has used. It terminates when the consumption of the CPU time has reached the specified limit.

In order to generate different load conditions on each LD host, we have also implemented an artificial load generator. The load generator can be run on each host to generate job requests either directly to local OS or to the DSF server through UA command line interface. When using the load generator, a mean job service time, a mean inter-arrival time and a number of generated jobs are specified. The service time of each job is generated from an exponential distribution with specified mean service time. After each job generation, the generator will wait for a period of time which is drawn from an exponential distribution with specified mean inter-arrival time. The load generator in each host is also responsible for collecting statistical information. For each job, the load generator records its start and termination times. The response time of each job is accumulated. At the end of each experiment, the mean response time, standard deviation, maximum and minimum response times, and average load value are reported by the load generator.

For the measurement purpose, we chose the mean response time as our performance measure for different LD algorithms. Different offered system load ρ can be adjusted by fixing the mean job service time $1/\mu$ and varying the mean inter-arrival time $1/\lambda$.

In the following experiments, we used 4 seconds CPU time as the mean job service time. The artificial load generator was run in each host with specified load value to generate 100 jobs. There were totally 300 jobs generated in each experiment. The number of active Run Job units allowed in the DSF server was set to 30 in order to avoid any queue formation at that unit as this undesirable situation seriously affects job's response time.

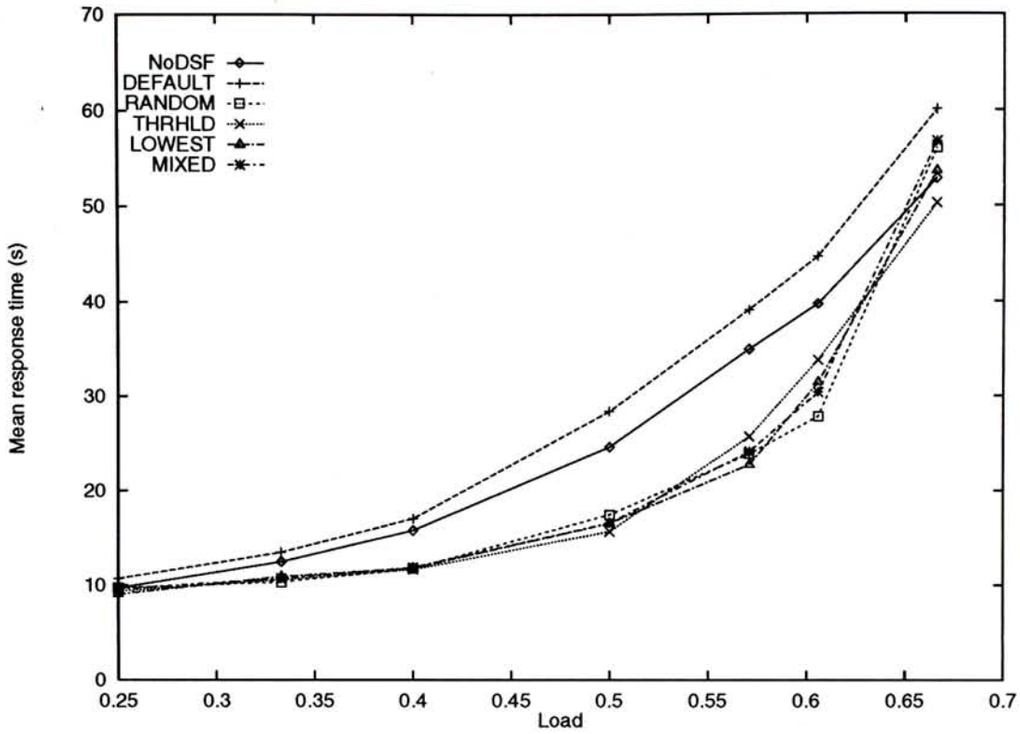

Figure 5.1: Mean response time versus offered system load in homogeneous workload condition for various LD algorithms.

5.3 Experimental results

5.3.1 Performance of LD algorithms

In this section, we study the performance of all LD algorithms under both homogeneous and heterogeneous workload conditions. For the homogeneous workload, the load value generated in different hosts is the same. However, the load values 0.33, 0.5 and 0.8 were generated in three different hosts respectively under heterogeneous workload condition. The load average value of 1.5 was used as the threshold in the transfer policy for all LD algorithms.

Figure 5.1 shows the mean response time under a range of offered system loads for various LD algorithms. The mean response time of DEFAULT is the largest among all LD algorithms. DEFAULT's growing rate is similar to NoDSF except that its mean response time is always larger. The difference between these two curves indicates precisely the overhead introduced by our DSF when DSF is solely used for local execution. In fact, providing the default algorithm for job submission is mainly for flexibility rather than any performance improvement. Its poorer performance than without using DSF is expected.

It should be noted that LD can improve system performance even each host in a distributed system is subjected to the same average workload. This is because LD can take advantage of those instantaneous load imbalance conditions and act accordingly by using the task transfer mechanism. As a result, the mean response time of the four LD algorithms is generally smaller than NoDSF except when the load level is above 0.6. At such system load level, only THRHLD is slightly below NoDSF and all remaining three algorithms have mean response time above it. This result is contrary to other findings. We believe that the following reasons account for such outcome. Firstly, we should note that the LD algorithms in our study are all sender-initiated algorithms and high system load condition is unfavorable to sender-initiated algorithm [9]. Secondly, we also observed that the overhead in using our DSF seems to be increasing when system load is increased. It may be due to our extensive using of threads inside the DSF server that hits some system limit. Hence, the benefit offered by LD under high system load is completely offset by the overhead incurred in our system. Thirdly, the Mach OS at system load above 0.6 starts to be unstable as system panics become more frequent. In particular, we met a variety of problems resulted from the UNIX server that makes any experiments for system load level above 0.66 meaningless. For the system load below 0.6, the performance gain from LD is small at low system load level but it increases gradually as system load is increased until system load has reached 0.57. This shows that LD resulted from sender-initiated algorithm is more effective when the system load is moderate. When system load is high, it should be more desirable to stop any LD activities resulted from using sender-initiated algorithm.

For the four LD algorithms under evaluation, their performance difference is not significant up to system load 0.4. At 0.5 load level, we find that THRHLD is the best among others but it becomes the worst at 0.57 and 0.6 load level. We intuitively expect that our THRHLD algorithm implementation will probably give the poorer performance than RANDOM and LOWEST because of the following reasons:

- THRHLD has acceptance policy. Hence, whenever a task transfer request is received by MM from peer MM, it needs to communicate with local THRHLD algorithm for getting acceptance decision. When the algorithm is asked for decision, it needs to further get the local system load for making its decision. This kind of overhead does not exist in RANDOM and LOWEST as their acceptance policy always accepts task transfer.

- Because of having acceptance decision, THRHLD is likely to make a bad decision which results in job re-submission to it again. The consequence is that the algorithm needs to do extra work in selecting another host from the list of LD hosts.

The job re-submission situation will occur very frequent when system load is increased. This is because the probability of finding an underloaded host decreases as the system load becomes high.

When RANDOM is considered, we should note that RANDOM in our implementation does not have lower overhead than THRHLD and LOWEST. The difference of overhead among RANDOM, THRHLD and LOWEST is usually resulted from the fact that RANDOM does not need to collect any load information but this is not true in our implementation. It is because LIM inside DSF server is regularly collecting and disseminating load information among LD hosts. We made this design decision because DSF is intended to support running multiple LD algorithms. The benefit of having one load information collection and dissemination mechanism will be realized when we are having two or more LD algorithms that are also using this information.

The performance of LOWEST is only between THRHLD and RANDOM except when load level is at 0.57. When considering MIXED algorithm, its mean response time is found to be among those resulted from RANDOM, THRHLD and LOWEST. Since each algorithm is having the same chance of being selected, it is natural to have an "averaged" result.

Figure 5.2 gives the plot of standard deviation of response time for all LD algorithms versus various system load levels. We can observe that LD can lower the variation of job's response time significantly when comparing a system without LD. As a result, the response time of jobs becomes more predictable and this is desirable to a real time system. When comparing with Figure 5.1, we notice that they are having a very similar pattern. When mean response time of jobs is low, the corresponding standard deviation of their response time is also low. This implies that mean response time and its standard deviation are highly correlated.

Table 5.1 gives the result of performance of different LD algorithms under heterogeneous workload condition. Both mean response time and standard deviation of response time are significantly improved under heterogeneous workload with respect to the system without LD. As expected, the system performance improvement is also much larger than that in the homogeneous workload condition. Under heterogeneous workload, the

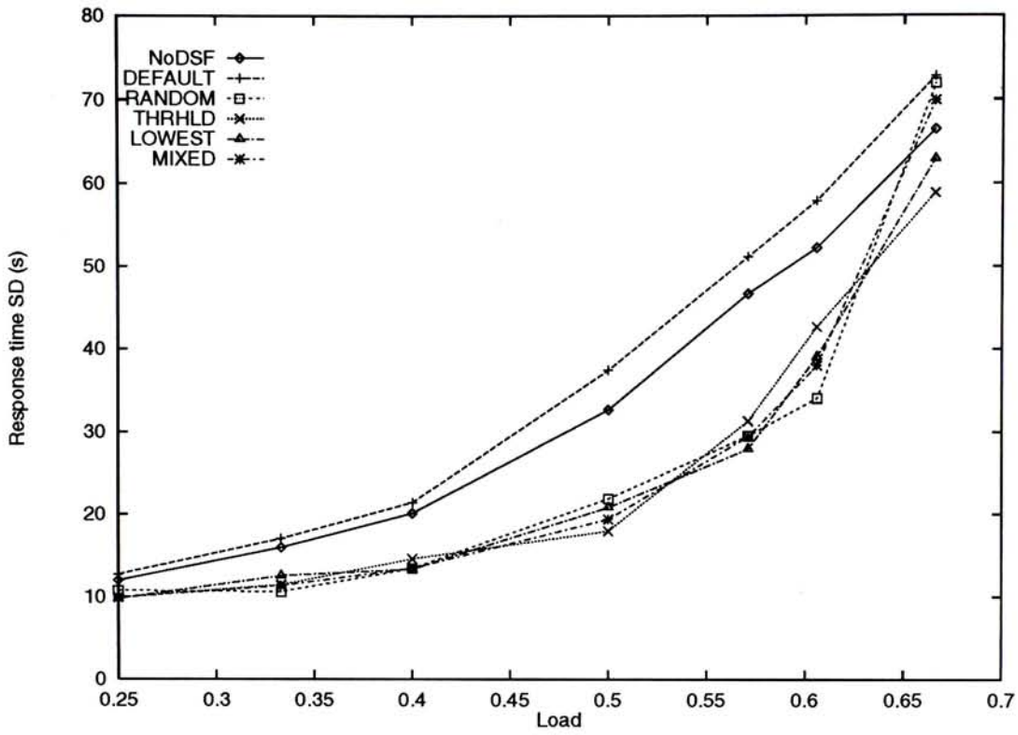

Figure 5.2: Standard deviation of response time versus offered system load in homogeneous workload condition for various LD algorithms.

LD	Response time (s)		Improvement %	
Algorithm	Mean	SD	Mean	SD
NoDSF	50.3	84.7	-	-
RANDOM	17.9	24.1	64.4	71.6
THRHLD	19.0	28.7	62.2	66.1
LOW	18.1	24.5	64.1	71.0
MIX	17.9	23.8	64.5	71.9

Table 5.1: The performance of different LD algorithms under heterogeneous workload condition.

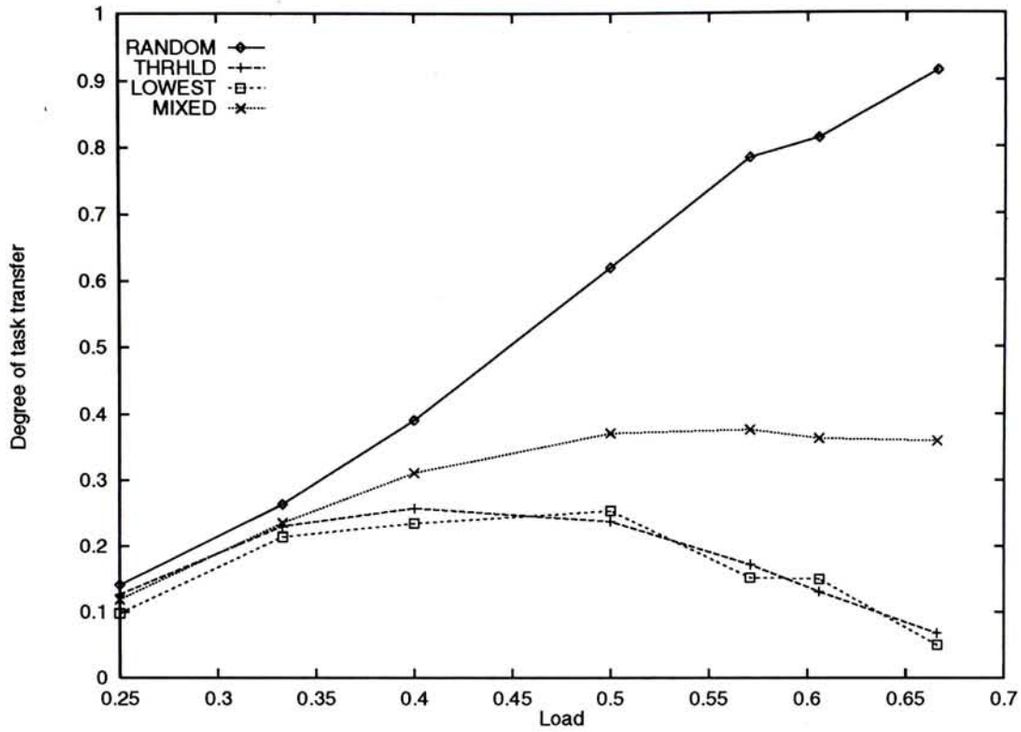

Figure 5.3: The degree of task transfer versus offered system load under homogeneous workload condition for various LD algorithms.

constant load imbalance condition creates a lot of opportunities for LD algorithm to redistribute the workload. Among the four LD algorithms, the improvement resulted from THRHLD is the worst because of the overhead of invoking its acceptance policy.

In summary, both mean response time and its standard deviation are improved with respect to the same system without LD under homogeneous workload condition. Also we showed that the mean response time and its standard deviation are highly correlated as they have a very similar performance curves. The result also shows that LD by using sender-initiated algorithms under homogeneous system load condition is desirable when system load is low to moderate. At high system load, it is more appropriate to suspend LD activities. However, LD can significantly improve system performance under heterogeneous workload condition. In addition, we also compared the performance between the condition of running multiple LD algorithms with its component algorithms. We got generally averaged performance from its component algorithms when using multiple LD algorithms for distributed scheduling.

5.3.2 Degree of task transfer

Figure 5.3 is the plot of degree of task transfer under different system loads for

various LD algorithms. The degree of task transfer is defined as the ratio of the number of transferred tasks to remote hosts to the number of tasks generated to local host. The degree of task transfer for RANDOM increases as the system load is increased. In contrast, THRHLD and LOWEST increase their degree of task transfer up to load level 0.4 and 0.5 respectively. After that, their degree of task transfer decreases as the system load is increased. This precisely reflects the difference when load information of other hosts is considered during making task transfer decision. Since RANDOM does not consider load information in its location policy, it tends to transfer more jobs to remote hosts under high system load because most of the time its local load is well above its threshold value used in transfer policy. Under high system load, the chance of finding an underloaded host in the system with homogeneous workload is very low. Thus both THRHLD and LOWEST will tend to execute the tasks locally because local MM will almost always receive REFUSE during task transfer negotiation for THRHLD, and the host selected by LOWEST in its location policy will always have load value well above its threshold value. The degree of task transfer for MIXED increases up to load level of 0.5 and then maintains at such level when load level is further increased. This is resulted from the balanced effect contributed by RANDOM which tends to increase the degree of task transfer, and LOWEST and THRHLD which tend to lower the task transfer degree under high system load.

5.3.3 Effect of threshold value

Figure 5.4 illustrates the performance difference of THRHLD when three different threshold values T (1.0, 1.5, 3.0) are used respectively. Although all the three LD algorithms are using threshold as their transfer policy, we chose THRHLD for our study because threshold value is also used in its acceptance policy which reflects the load condition of remote host. The experiment was carried out under homogeneous workload condition.

From the figure, we can recognize that threshold value 1.5 is the optimum for THRHLD when the system load is between 0.25 to 0.53. Above load level 0.53, threshold value 3 can give further system performance improvement. Thus it is more appropriate to have high threshold value for THRHLD when system load is increased. Intuitively, we should expect that when system load is low, a smaller threshold value should give a better performance. However, we failed to demonstrate that the system performance can be improved with respect to $T = 1.5$ when threshold value 1 is used. We believe that

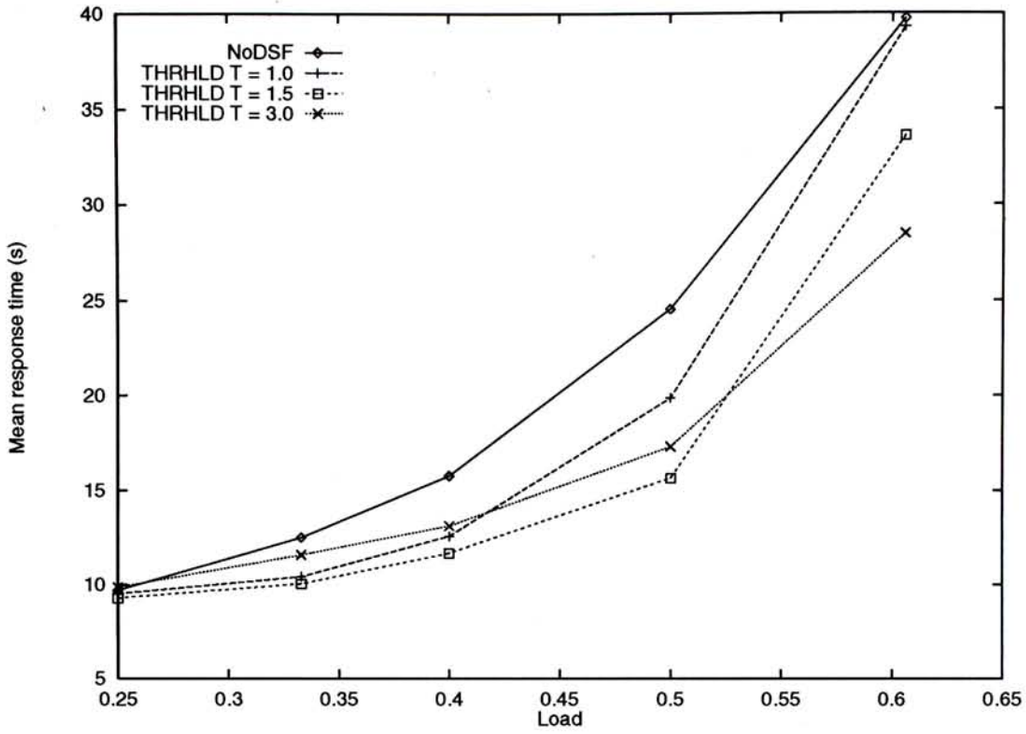

Figure 5.4: Effect of threshold value on performance of THRHLD algorithm.

if another optimal threshold value under low system load does exist, its value should fall between 1 to 1.5.

Chapter 6

Conclusion and Future Work

6.1 Summary and Conclusion

A LD facility is required to realize the potential benefits brought by load distribution in a distributed system. In this thesis, we have presented the design issues and the implementation details of the Distributed Scheduling Framework (DSF) which is a LD tool running on the Mach operating system. DSF consists of three components—the DSF server, the LD library and the User-Agent. DSF is designed to separate policies from mechanisms in load distribution. Thus this results in three logically separated modules (LIM, MM, DM) in the DSF server. The architecture of DSF was described and the design issues and functionalities of different DSF components were discussed. In particular, DSF has the following features:

- It supports distributed scheduling by using removable LD algorithms. The replacement of LD algorithms can be done conveniently at the user level without shutting down the system or any DSF servers.
- It allows multiple LD algorithms running concurrently on the same host to provide distributed scheduling.
- It facilitates the implementation of LD algorithms by providing LD library to algorithm writers.
- The number of hosts in DSF system can be changed dynamically and therefore any host can join and leave the system without requiring to terminate the whole system first.

We have done a preliminary study of some well-known LD algorithms by using our DSF. The result shows that LD by using sender-initiated algorithms under homogeneous system load condition is desirable when system load is low to moderate. At high system load, it is more appropriate to suspend LD activities. However, LD can significantly improve system performance under heterogeneous workload condition. We got generally averaged performance from component algorithms when using multiple LD algorithms for distributed scheduling.

6.2 Future Work

DSF currently only supports sender-initiated LD algorithms. In order to support receiver-initiated LD algorithms, we need to make several modifications on DSF server and the stub function. Our modifications assume that only one receiver-initiated algorithm is running at a time but it should not be difficult to extend the modifications to support running multiple receiver-initiated algorithms.

When Run Job unit in MM has finished job execution, it should be modified to notify the registered receiver-initiated algorithm for such change of status. We need to modify the stub function so that the algorithm can support receiving such message. The algorithm then checks the system load of local host. If its value is below some threshold value (transfer policy), it tries to negotiate with the corresponding algorithm in remote host for making job reservation so that a job will be transferred to local host when the job is submitted to that algorithm in remote host next time. The negotiation between the algorithms is achieved by using the send-and-receive communication service provided by LD library.

When a job is submitted to the algorithm, it checks if it has job reservation for an algorithm in remote host. If not, the job is executed locally. Otherwise, it transfers the submitted job to the remote host that has previously made the job reservation. The algorithm should neglect a job reservation that has been made before some period (expiration period) as the remote host may no longer be in underloaded state.

In the current design, we observe that there is a socket message server in each module inside the DSF server. The Message Server in LIM and MM is an iterative server but the Socket Message Server in DM is a concurrent one. We believe that it is possible to eliminate all these socket message servers which are scattered in each module and provides a centralized concurrent server inside the DSF server to handle all incoming

message requests. As a result, the redundant code can be eliminated. However, we need to be cautious in determining the maximum number of allowed network connections. Since each connection will consume one file descriptor, it will be very easy to use up all available file descriptors allowed for a given task if the value is set to too large. If the number is set to too small, the centralized message server may become a bottleneck in message processing.

In Chapter 4, we pointed out the problem of using non-reentrant library functions in multithreaded programming. This problem may make it impossible to write a program that is completely thread-safe. In addition, the problem of frequent panics under high system load during experimental studies also limits DSF as being a good experimental tool because conducting experiment by submitting a large number of jobs is nearly impossible. Since CMU has stopped its research and development work in Mach, it is unlikely that the above problems will be solved shortly. The possible way to overcome the above problems is to port DSF to other modern UNIX systems, like Solaris, IRIX, that have proper support of multithreaded programming. Since POSIX threads (Pthreads) is becoming a standard on UNIX systems, it can be used as a replacement of cthreads library which is provided in Mach. Moreover, this can also provide an opportunity for making DSF to perform LD in heterogeneous computing environment. Therefore, most of the effort in porting DSF will include the following two areas:

- Replace calling cthreads library interface by the corresponding pthread library interface.
- Replace all Mach IPC by socket IPC.

The first area is more demanding but the second could be accomplished easily as Mach IPC at current stage is only restricted to communication between DM and LD algorithms.

Appendix A

LD Library

The LD library facilitates the implementation of LD algorithms by providing the following functions:

- `alg_gethostinfo()`
- `alg_gethostalg()`
- `alg_transfer()`
- `alg_msg()`

to LD algorithm writers. The manual page of each function is given as follows:

NAME

alg_gethostinfo - get a list of hosts participating in LD scheme

SYNOPSIS

```
#include <algo.h>

int alg_gethostinfo(void *host_buf, int host_buf_size);
```

DESCRIPTION

The `alg_gethostinfo()` function puts a list of hosts which are currently participating in LD in `host_buf` with their current load information. `host_buf` is a buffer of size `host_buf_size` for storing returned host information. It should be defined as an array of `struct h_info` structure which is defined in `"../lim/lim.h"`:

```
struct h_info {
  char name[MAXHOSTNAMELEN];
  struct in_addr hostid;
  struct load_info load;
};
```

RETURN VALUES

On success, the number indicates the number of hosts available. On error, a negative number is returned.

-1

Error in making Mach RPC to DM.

-2

`host_buf` is too small for storing all returned host information.

NAME

alg_gethostalg - get a list of loaded LD algorithm descriptions for a particular host

SYNOPSIS

```
#include <algo.h>

int alg_gethostalgo(char *host, void *algo_buf, int algo_buf_size);
```

DESCRIPTION

The `alg_gethostalgo()` function puts a list of loaded LD algorithms in `algo_buf` for the specified host `host`. `algo_buf` is a buffer of size `algo_buf_size` for storing algorithm information. It should be defined as an array of `struct algo_info` structure which is defined in `"../dm/dm.h"`:

```
struct algo_info {
  short id; /* algo identity */
  char desc[50]; /* algo description */
  int action; /* default acceptance decision */
  int algo_pass_lim; /* max no. of times that a given job can
                        be submitted to algorithm */
  u_short p_num; /* port number */
};
```

RETURN VALUES

On success, the number indicates the number of algorithms loaded in `host`. On error, a negative number is returned.

-1

Error in making Mach RPC to DM.

-2

`algo_buf` is too small for storing all returned algorithm information.

NAME

alg_transfer - submit a job to local MM for task transfer negotiation or execution

SYNOPSIS

```
#include <algo.h>

void alg_transfer(struct job_ent *job);
```

DESCRIPTION

The `struct job_ent` structure is defined in "`.../mm/mm.h`":

```
struct job_ent {
  short src_algoid, /* source algo id */
        tar_algoid; /* target algo id */
  char cmd[CMDLEN], /* command line */
       h_ori[MAXHOSTNAMELEN], /* originated host */
       h_tar[MAXHOSTNAMELEN]; /* target host */
  int ua_job_id; /* job id in originated UA */
  u_short ua_port;
  struct {
    uid_t uid;
    gid_t gid;
    char cwd[MAXPATHLEN];
  } job_env;
  int algo_pass; /* no. of times passed to algorithm
                    in a given host */
  struct job_ent *next;
};
```

It contains all the necessary information about a submitted job including the target host `h_tar` for job execution. If `h_tar` is a remote host, a task transfer will be carried out by the local MM.

NOTES

`job` should be a pointer to `struct job_ent` structure but not an address of such structure as the memory storage pointed by `job` will be released after invoking this function.

NAME

alg_msg - send a message to a LD algorithm on a host using send-and-receive communication model

SYNOPSIS

```
#include <algo.h>

int alg_msg(char *host, short algo_id, void *msg);
```

DESCRIPTION

The `alg_msg()` function sends a message `msg` to LD algorithm `algo_id` in host `host`. `msg` is a message buffer of 1024 bytes. It is used for holding sending message content. When invoking this function is successful, it holds the content of reply message from algorithm `algo_id`.

RETURN VALUES

On success, a zero value is returned. On error, -1 is returned.

Appendix B

Sample Implementation of LD algorithms

In this appendix, we give our sample implementation of some LD algorithms described in Section 5.1.

B.1 LOWEST

```
 1 /*
 2  * Transfer policy: threshold
 3  * Location policy: lowest
 4  */
 5 #include <string.h>
 6 #include "algo.h"
 7
 8 #ifndef THRES
 9 #define THRES 2 /* Threshold for transfer policy */
10 #endif /* THRES */
11
12 /* function prototypes */
13 void alg_init(void);
14 void submit_job(struct job_ent *);
15
16 void
17 alg_init(void)
18 {
19   /* fill in registration */
20   a_info.id = 10;
21   strcpy(a_info.desc, "Threshold transfer Lowest location");
22   a_info.action = ACCEPT;
```

```
23
24    submitjob_fp = submit_job;
25    /* consult_fp = consult; */
26
27    /* algorithm specific initialization */
28  }
29  /* end of alg_init() */
30
31  void
32  submit_job(struct job_ent *job)
33  {
34    struct h_info h_list[8];
35    int hnum; /* no. of hosts available */
36    int choice;
37
38    hnum = alg_gethostinfo(h_list, sizeof(h_list));
39
40    /* Threshold transfer policy */
41    /* h_list[0] is local host */
42    if (h_list[0].load.avenrun[0] < THRES) /* local processing */
43      choice = 0;
44    else { /* remote processing */
45      int i,
46          lowest = 0;
47
48      /* Lowest location policy */
49      for (i = 1; i < hnum; i++) {
50        if (h_list[i].load.avenrun[0] <
51            h_list[lowest].load.avenrun[0])
52          lowest = i;
53      }
54      if (h_list[lowest].load.avenrun[0] < THRES)
55        choice = lowest;
56      else
57        choice = 0;
58    }
59    strcpy(job->h_tar, h_list[choice].name);
60
61    alg_transfer(job);
62  }
63  /* end of submit_job() */
```

In LOWEST, we only have to provide two functions `alg_init()` and `submit_job()`. The function of `alg_init()` is to fill in the information about the algorithm in variable `a_info` which is declared in `algo.h`. Since the algorithm indicates that it always accepts

remote task transfer request (line 22), we do not need to provide a consult function in line 25.

`submit_job()` is invoked when there is a job submitted by DM. It first gets the load information of all LD hosts by calling the LD library function `alg_gethostinfo()` (line 38). Line 42 to 44 is the threshold policy. If the local load is above the threshold value, the location policy is invoked (line 45 to 58). LOWEST just picks the host with the lowest load from its list (line 49 to 53) and makes it as the choice if its value is also below the threshold value (line 54 to 57). Finally, it invokes the LD library function `alg_transfer()` (line 61) to send the job to MM for either task transfer or local execution.

B.2 THRHLD

```
 1 /*
 2  * Transfer policy: threshold
 3  * Location policy: threshold
 4  */
 5 #include <string.h>
 6 #include <stdlib.h> /* for rand() */
 7 #include <time.h>
 8 #include "algo.h"
 9
10 #ifndef THRES
11 #define THRES 2 /* Threshold for transfer policy */
12 #endif /* THRES */
13
14 /* function prototypes */
15 void alg_init(void);
16 void submit_job(struct job_ent *);
17 int consult(void);
18
19 void
20 alg_init(void)
21 {
22   /* fill in registration info */
23   a_info.id = 5;
24   strcpy(a_info.desc, "Threshold transfer & location");
25   a_info.action = CONSULT;
26
27   submitjob_fp = submit_job;
28   consult_fp = consult;
29
```

```
30   /* algorithm specific initialization */
31   srand(time(NULL));
32 }
33 /* end of alg_init() */
34
35 void
36 submit_job(struct job_ent *job)
37 {
38   struct h_info h_list[8];
39   int hnum; /* no. of hosts available */
40   int choice;
41
42   hnum = alg_gethostinfo(h_list, sizeof(h_list));
43
44   if (job->algo_pass > 0) { /* previous bad decision */
45     if (hnum == 1)
46       choice = 0;
47     else { /* hnum > 1 */
48       int count = 0;
49
50       /* Try to find another host which isn't the same
51          as previous decision.  Give up if trial limit
52          has reached */
53       for (;;) {
54         choice = rand() % (hnum - 1);
55         choice++;
56
57         /* we've got different host from previous one */
58         if (strcmp(h_list[choice].name, job->h_tar) != 0)
59           break;
60
61         if (++count == 5) {
62           /* limit has reached, resort to local host */
63           choice = 0;
64           break;
65         }
66       }
67     }
68   }
69   else { /* first time submit to algo */
70     /* Threshold transfer policy */
71     if (h_list[0].load.avenrun[0] < THRES) /* local processing */
72       choice = 0;
73     else { /* remote processing */
74       if (hnum == 1)
75         choice = 0;
```

```
76        else {
77          choice = rand() % (hnum - 1);
78          choice++;
79        }
80      }
81    }
82    strcpy(job->h_tar, h_list[choice].name);
83
84    alg_transfer(job);
85  }
86  /* end of submitjob() */
87
88  int
89  consult(void)
90  {
91    struct h_info h_list[8];
92    int hnum;
93
94    hnum = alg_gethostinfo(h_list, sizeof(h_list));
95
96    /* Threshold location policy */
97    if (h_list[0].load.avenrun[0] < THRES)
98      return ACCEPT;
99    else
100     return REJECT;
101 }
102 /* end of consult() */
```

THRHLD is different from LOWEST in that its default decision to task transfer request from remote host is CONSULT (line 25), thus it also needs to provide the `consult()` function (line 88 to 101).

`consult()` is invoked when MM has received task transfer request from peer MM with the job information specifying THRHLD as target algorithm. It calls LD library function `alg_gethostinfo()` to get local load information since this information is in the first member of the returned list. Then it makes its decision on accepting or refusing the request based on the comparison between local load value with threshold value (line 97 to 100).

In `submit_job()`, the algorithm determines whether the job is submitted to it the first time or more (line 44 to 81). For the first time (line 69 to 81), it compares its load with threshold value. If it is above the threshold value, it randomly selects a host for transferring the job. Otherwise, local host will be chosen for execution. For the case of more than one time (line 44 to 68), it tries to find another host that is different from

previous one (line 53 to 66). If this process has failed for more than five times, a local host is chosen (line 61 to 65).

Appendix C

Installation Guide

C.1 Software Requirement

In order to compile the DSF package, the following programs are needed:

- GNU GCC
- GNU make (gmake)

They should be found in `/usr/mach3/bin` in a properly installed CMU Mach 3 system. However, since the above programs provided by the native CMU Mach 3 system are quite old, we suggest to install the latest version of them in `/usr/local/bin`. An X client program `xless` is also required by UA in the DSF package. It can be found in the following site:

```
ftp://ftp.x.org/pub/contrib/applications
```

In addition, the following programs are required in extracting the DSF package:

- GNU tar (gtar)
- GNU gzip

The latest version of them can be found in any GNU ftp site, such as `prep.ai.mit.edu`. In the following section, we assume that the latest version of the above mentioned programs have already been installed. In particular, we use the latest version of gcc from `/usr/local/bin` but not the one from `/usr/mach3/bin`.

C.2 Installation Steps

1. The DSF package is available as a gzip'd tar file `dsf.tar.gz`. It can be extracted under `/usr/local` directory by the following steps:

   ```
   cd /usr/local
   gtar xfvz .../dsf.tar.gz
   ```

   The source tree of DSF should be created under `/usr/local/dsf` directory. The following subdirectories `lim`, `mm`, `dm`, `algo`, `ua`, `ualgo` and `exp` should be found in `/usr/local/dsf`. `lim`, `mm` and `dm` contain the source for LIM, MM and DM respectively. `algo` and `ua` contain the source for LD library and UA respectively. `ualgo` contains the sample implementation of some LD algorithms and `exp` contains the tools for conducting LD experiments.

2. Before using make to compile the whole package, some environment variables should be set. Assuming you are using C shell from CMU Mach system, you type the following:

   ```
   setpath CPATH -i0 /usr/mach3/include
   setpath LPATH -i /usr/mach3/lib:/lib:/usr/lib
   ```

3. Then compile the DSF package by:

   ```
   cd /usr/local/dsf
   gmake
   ```

   The compilation steps performed by `gmake` include the following:

   (a) it goes to `lim` directory and compiles the library `liblim.a`.

   (b) it goes to `dm` directory and compiles two libraries `libdm.a` and `libdsfdm.a`. `libdm.a` is used by DSF server and `libdsfdm.a` is used by LD algorithms.

   (c) it goes to `mm` directory and compiles the library `libmm.a`.

   (d) it goes to `algo` directory and compiles `libalgo.a` and `libdsfalgo.a`. The library `libalgo.a` is used by DSF server and the LD library `libdsfalgo.a` is used by LD algorithms.

   (e) the DSF server is then compiled.

   (f) it goes to `ua` directory and compiles UA program.

(g) it goes to **ualgo** directory and compiles the sample LD algorithms.

(h) it goes to **exp** directory and compiles `job`, `lg` and `lg2`. `job` is the artificial task. `lg` is the artificial load generator. `lg2` is the artificial load generator for running multiple LD algorithms experiment.

When the compilation is finished, the following executables should be found:

- the DSF server (`dsfd`)
- the UA (`ua/ua`)

The following files are needed for compiling any new LD algorithm:

- LD library header file (`algo/algo.h`)
- LD and supporting libraries:
  - `algo/libdsfalgo.a`
  - `dm/libdsfdm.a`
- LD algorithm stub (`algo/algo_stub.o`)

C.3 Configuration

The following macros in `/usr/local/dsf/common.h` can be changed before compilation to modify the default behavior of the DSF server:

- `CHK_PER` is the interval used by the Check Algo unit in DM to determine how frequent to check the status of all loaded LD algorithms.
- `SYS_APASS_LIM` is the default number of times that a job can be resubmitted to the same LD algorithm.
- `JEXEC_MAX` is the maximum number of active Run Job unit allowed in MM.

Appendix D

User's Guide

D.1 The DSF server

The DSF server can be started by running the following command:

```
dsfd &
```

If no options are given, the DSF server will use event driven method to exchange load information. To start the DSF server using periodic load exchange method, add the `-p` option as follows:

```
dsfd -p &
```

By default, the maximum number of active Run Job unit in MM is set to 30. It can be changed to other value when starting the DSF server as follows:

```
dsfd -j35 &
```

D.2 The User Agent

The User Agent (UA) allows job submission either at the command line or interactively. The UA is in interactive mode when it is started without specifying any options as follows:

```
ua
ua>
```

A help message of supported commands can be obtained by typing `help` command at the UA prompt

```
ua> h
Commands:
quit (q)        Quit UA
help (h)        Show this message
batch (b)       Submit batch jobs
        batch jobfile
submit (s)      Submit a job
        submit command:source LD algorithm:target LD algorithm
readlog (rl)    Read job output log
jobstat (js)    Display job status
dsfstat (ds)    Display DSF status
ua>
```

To obtain the number of LD hosts in a DSF system and the type of LD algorithms currently loaded in each host, we use the **dsfstat** command

```
ua> ds
Host: para202
Load average: 0.00, 0.00, 0.00  Mach factor: 1.00, 0.97, 0.94
Loaded LD algorithm
ID ACTION   DESCRIPTION
 0 Accept   Default algorithm
 1 Accept   Threshold transfer Random location

Host: para199
Load average: 0.00, 0.00, 0.00  Mach factor: 1.00, 0.97, 0.94
Loaded LD algorithm
ID ACTION   DESCRIPTION
 0 Accept   Default algorithm
 1 Accept   Threshold transfer Random location

Host: para197
Load average: 0.00, 0.00, 0.00  Mach factor: 1.00, 0.97, 0.94
Loaded LD algorithm
ID ACTION   DESCRIPTION
 0 Accept   Default algorithm
 1 Accept   Threshold transfer Random location

Total no. of LD host(s): 3
```

To submit a job, we use the **submit** command as follows:

```
ua> s ms -h:1:1
```

We can get the status of currently submitted job(s) by using **jobstat** command

```
ua> js
 JID STAT R_HOST          COMMAND
   3 R    para202         ms -h
```

When the job is finished, UA will report this as follows:

```
[3] done    ms -h
```

The output of the command is stored in a temporary file. It can be examined by using **readlog** command

```
ua> rl
Host 2, default proc. set 3, default proc. set name 4,
default pager 20
I386 AT, 1 of 1 cpus available, 16.0M memory
10000us minimum timeout, 10000us minimum quantum
Load averages:  0.00  0.00  0.00
Mach factors:   1.00  0.97  0.94
Mach_3.0 VERSION(MK83): Sun Jan  5 17:24:57 HKT 1997; kernel/STD+WS
(para202.cs.cuhk.edu.hk)
```

To submit a batch job, a job file is needed. This file contains lines of job specification which is identical to the one specified in **submit** command. Blank lines and lines starting with # character are ignored.

The following is an example of submitting a job at the command line without entering UA interactive mode:

```
ua 'ms -h:1:1'
```

The command line argument is the same as the one specified in **submit** command when UA is in interactive mode. To submit a batch job at the command line, we type the following:

```
ua -f batch_file
```

To select **xless** (an X client program) for displaying output of each submitted job, we specify **-x** option before starting UA

```
ua -x
```

D.3 LD experiment

Before conducting LD experiment, the DSF server should have been started and the relevant LD algorithms should have been loaded. For example, if we want to conduct an experiment on the LD algorithm (identity 1), we use the artificial load generator to start the experiment as follows:

```
lg -s344 -l1 4 8 100
```

The `-s` option is the initial seed value. The `-l` option specifies the algorithm identity to which we are going to submit the generated job. The remaining three numbers represent the mean job service time (in second) , the mean job inter-arrival time (in second) and the number of generated jobs.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, 1986.

[2] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX distributed operating system: load balancing for UNIX.* Springer-verlag, 1993.

[3] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures*, pages 11–30, 1992.

[4] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach.* Addison-Wesley, 1993.

[5] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Feb. 1988.

[6] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, second edition, 1994.

[7] P. Dikshit, S. K. Tripathi, and P. Jalote. SAHAYOG: A test bed for evaluating dynamic load-sharing policies. *Software—Practice and Experience*, 19(5):411–435, May 1989.

[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[10] G. Gold and T. Schnekenburger. Using the ALDY load distribution system for PVM applications. In *Parallel Virtual Machine-EuroPVM'96 Third European PVM Conference Proceedings*, pages 278–287. Springer-Verlag, Oct. 1996.

[11] D. Julin. Network server design, mach networking group. Unpublished report, Sept. 1989.

[12] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *Proceedings of the 8th International Conference on Distribued Computer Systems*, pages 123–130, June 1988.

[13] S. J. Leffler, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4bsd interprocess communication tutorial. In *UNIX Programmer's Supplementary Documents (PSD) 4.4 Berkeley Software Distribution*, page PSD:21. Computer Systems Research Group, EECS, U.C. Berkeley, June 1993.

[14] K. Loepere. *Mach 3 Kernel Interface.* Open Software Foundation and Carnegie Mellon University, July 1992.

[15] K. Loepere. *Mach 3 Kernel Principles.* Open Software Foundation and Carnegie Mellon University, July 1992.

[16] K. Loepere. *Mach 3 Server Writer's Guide.* Open Software Foundation and Carnegie Mellon University, July 1992.

[17] C. Lu and S. L. Hsieh. Facilitating load distribution based on microkernel technology. In *Proceedings of International Conference On Signal Processing Applications & Technology*, volume 1, pages 776–780, Oct. 1996.

[18] C. Lu, J. C. S. Lui, P. W. K. Lie, M. K. Tang, S. Y. Lau, and H. K. Li. Distributed scheduling framework — a load distribution facility on mach. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 757–768, 1996.

[19] Q. Lu and A. S. L. Hsieh. DSF: a load distribution facility supporting multiple algorithms on mach. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 1, pages 488–497, June 1997.

[20] D. S. Milojicic. *Load Distribution: Implementation for the Mach Microkernel.* Verlag Vieweg, 1994.

[21] D. S. Milojicic, P. Giese, and W. Zint. Experiences with load distribution on top of the mach microkernel. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 19–36, Sept. 1993.

[22] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Society International Conference COMPCON 89*, Feb. 1989.

[23] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25:33–44, Dec. 1992.

[24] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison-Wesley, fourth edition, 1994.

[25] W. R. Stevens. *UNIX Network Programming.* Prentice-Hall, 1991.

[26] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[27] S. R. Tsai, J.-T. Chiou, and H.-T. Jen. Load balance facility in distributed MINIX system. In *Proceedings of the 20th EUROMICRO Conference. EUROMICRO 94. System Architecture and Integration*, pages 162–169, Sept. 1994.

[28] Y.-T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, Mar. 1985.

[29] S. Zhou. An experimental assessment of resource queue lengths as load indices. Technical Report UCB/CSD 86/298, Computer Science Division (EECS), University of California, Berkeley, June 1986.

[30] S. Zhou. Performance studies of dynamic load balancing in distributed systems. Technical Report CSD-87-376, University of California, Berkeley, 1987. PhD thesis.

[31] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, Sept. 1988.

[32] S. Zhou and D. Ferrari. A measurement study of load balancing performance. In *Proceedngs of 7th International Conference on Distributed Computing Systems*, pages 490–497, Oct. 1987.

[33] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software—Practice and Experience*, 23(12):1305–1336, Dec. 1993.

[34] W. Zhu. Dynamic load balancing on amoeba. In V. L. Narashimhan, editor, *Proceedings of IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, volume 1, pages 355–364. IEEE, Apr. 1995.

[35] W. Zhu and C. Steketee. An experimental study of load balancing on amoeba. In *Proceedings of The First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, pages 220–226. IEEE, Mar. 1995.